PARTICLE: An Automatic Program Specialization System for Imperative and Low-Level Languages

by

Nathaniel David Osgood

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

Massachusetts Institute of Technology

September 1993

Signature of Author ....
Department of Electrical Engineering and Computer Science
August 15, 1993

Certified by ....
Stephen A. Ward
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ....
Campbell L. Searle
Chair, Department Committee on Graduate Students

For My Dearest Friend,
Who Already Has the Soul of a Bodhisattva,
And Just Hasn't Realized It Yet.

Nate

Acknowledgements

The author wishes to thank a number of individuals for their help during the time at which this thesis was being designed, implemented, and described. The gratitude felt is deep but admits no ordering, and an individual's position within the list should not be taken as any indication of the degree of appreciation to be expressed.

Particular thanks are due to:

- **Steve Ward**, for agreeing to supervise this thesis, providing great insight into the structure of the problem domain, and for allowing me the freedom to pursue my ideas while always being ready to serve as a highly valuable source of advice and support.

- **My Parents**, for agreeing to supervise this life, providing great insight into the structure of the problem domain, and for allowing me the freedom to pursue my ideas while always being ready to serve as a highly valuable source of advice and support.

- **Shaun Kaneshiro**, for enormous support at untold numbers of points during the writing of this document, for discussing the ideas underlying this thesis immediately after their inception, and for punctuating grim daily thesis work with great intellectual stimulation, inspiration, and fun.

- **Chris Tsien**, for being a great friend on whom I can always rely, and for serving as a point of sanity in what has seemed like an insane world.

- **Ellen Spertus**, for serving as a great friend throughout the period in which this thesis was written, for unceasingly offering encouragement and enthusiasm for my work, for a number of insights into and suggestions for PARTICLE, and for an *enormously* valuable and detailed commentary on the early parts of this document.

- **Bjarne Steensgaard**, for wonderfully extensive and important commentary on the major portions of the thesis, and for enormous patience throughout the entire process – despite the fact that i am a total stranger to him.

- **Raymie Stata**, for a number of influential and insightful discussions of the ideas behind this thesis, for his continual interest in this work.

- **Shail Aditya**, for providing perceptive comments on partial evaluation and compilation technology, and for putting up with egregious lapses in the apartment cleaning schedule.

- **Harry Merrick**, for inviting me as a staff member to the Koobi Fora Field School and tolerating my frequent "zoning out" to think about my thesis. I have never found any more conducive an environment for careful thinking, and Harry's enormous patience and exceptional character made it a joy to work with the field school. Harry is also to be thanked enormously for putting up with my constant and no doubt infuriating alternation between thesis work and archeology projects. I am deeply grateful for Harry's contant support in pursuing my interdisciplinary interests.

- **Mark Smith**, for his enormous patience in presenting amazingly clear explanations of fascinating technical material at "wierd-ass" restaurants, and for his discussion of issues related to this thesis.

PARTICLE: An Automatic Program Specialization System for Imperative and Low-Level Languages

by

Nathaniel David Osgood

Submitted to the Department of Electrical Engineering and Computer Science on August 15, 1993, in partial fulfillment of the requirements for the degree of

Master of Science

in Electrical Engineering and Computer Science

Abstract

This document describes the design and implementation of PARTICLE, a fully automatic program specialization system for imperative and low-level languages. PARTICLE performs source-to-source specialization of programs written in the C programming language based on knowledge of program behavior acquired through extensible, high-quality program analysis. PARTICLE supports a unique two-phase specialization strategy that permits careful balancing between the time benefits and space costs of specialization and permits the user to avoid the huge space expansion that has traditionally accompanied program specialization. During the first phase of PARTICLE operation, an input program in the C programming language is analyzed using an extensibly precise form of symbolic evaluation in order to discover opportunities for low-level specialization. The symbolic evaluation manipulates explicit approximations to program values and states and the compile time flow of analysis in the program represents a close approximation to the run time flow of execution through that program. PARTICLE draws approximations to program values from a rich and extensible abstract domain, allowing the user to request arbitrarily precise approximations to program values in those regions of a program at which such precision is deemed desirable. PARTICLE analyzes program control flow in a framework that permits the system to simulate run time execution in as precise a manner as is desired by the user and possible given the system's knowledge about program data values and the need to guarantee specializer termination for all input programs. The precision of PARTICLE's control-flow simulation allows the system to discover and apply optimizations far outside the range of traditional compiler technology. In the second phase of PARTICLE operation, the system can make use of data collected during program analysis to explicitly weigh the performance benefits and space costs associated with different possible patterns of function-level specialization, given user time/space tradeoffs and global knowledge about program behavior. Following the choice of a desired pattern of specialization, the system outputs an appropriately specialized residual program. Preliminary performance results suggest that PARTICLE can offer substantial and sometimes dramatic performance boosts for certain classes of input programs. Unfortunately, PARTICLE suffers from extremely long running times and is limited to processing relatively small programs.

Thesis Supervisor: Stephen A. Ward
Title: Professor of Computer Science and Engineering

Contents

List of Figures

Chapter 1

Introduction

The use of abstraction and generality in software design offers a number of important benefits: Easy reuse, modifiability, and modularity of code. Adherence to such principles is widely viewed as highly important in the design and maintainance of large-scale software systems [24][38][53].

However, any software engineer concerned with the run time efficiency of his or her code must face the uncomfortable fact that while the consistent use of abstraction and generality in software development simplifies the software construction process, it can be associated with a substantial performance penalty. While the use of modular and clean, general interfaces in software improve *human* efficiency in designing software, frequently adherence to such techniques can decrease the *machine* efficiency of that software. As the size and complexity of software system grow and the use of abstraction and generality in software design expands, there is a growing need for techniques that lessen the performance cost of these abstraction mechanisms. One of the most promising methods for addressing this problem is automatic *program specialization.*

Program specialization is a program transformation technique that accepts as input a program and outputs a specialized (or "residual") program that has the same functionality as the original program, but exploits knowledge of characteristics of the data used in the original program to safely exploit optimized special cases where the original program made use of general, abstract mechanisms. Program specialization thus offers an automatic means of safely attaining the performance advantages of "hard coding" special cases without compromising the generality and abstract character of the original program. In many instances, the performance advantages gained through specialization are great, and can involve the elimination of complicated general-purpose mechanisms where they are not needed and the shifting of substantial computations from run time to compile time.

While existing optimizing compilers typically take significant advantage of specialization opportunities, the specializations performed by such systems are rather limited in scope and take place only at relatively low levels of program structure. Despite the high frequency of exact static knowledge of function parameters, the important opportunities for specializing functions based on common patterns of use throughout the program are not exploited by traditional systems. Moreover, in most existing commercial compilers, functions are treated as "firewalls" during analysis, so that even simple information about potentially crucial aspects of function behavior (or misbehavior) are not available to areas of the program that make use of this function. As a result, in current compilation systems abstraction barriers created primarily to foster human productivity and ease of comprehension not only become rigidly reified in the run time system but frequently also serve as major barriers to high quality program analysis. At an even higher level, existing compilation systems fail to take advantage of the fact that frequently knowledge of one portion of a program's *input data* is available is available prior to other components. In an application of the technique of *partial evaluation*, the use of partially specified input data can frequently allow the creation a program highly optimized to operate on a particular domain of input data with a dramatic increase in performance over the original program.

The use of partial evaluation as a practical means of program specialization has begun to be seriously investigated in the functional language community[5][6][56]. This work has documented enormous opportunities for speedup of abstractly written code in these languages[5], and suggests that partial evaluation represents a valuable addition to existing compiler techniques for such languages.

While truly practical approaches to automatic program specialization are still very much a research topic even for "laboratory" languages, preliminary studies have hinted at such tremendous advantages that it is natural to consider the adaptation of similar strategies for use in more traditional and popular imperative and low-level languages. The creation of effective specialization techniques for use in these domains has been regarded as a substantial open challenge for research[28]; while the operation of traditional languages often tends to be rather intuitively simple, the formal semantics of such languages is often far more complicated than for many functional languages, and the mechanisms that must be supported to correctly, efficiently, and thoroughly specialize programs in such languages can be correspondingly more complex. As a result, while functional languages can be processed by relatively simple partial evaluators [56], imperative languages demand processing that is substantially more involved. While achieving the goal of enabling practical, high-quality specialization of imperative and low-level languages represents a serious challenge, the rewards for doing so are potentially great: Such languages are extremely widely used, and are often employed in the most performance-critical applications. Because of their popularity and the accompanying "greed for speed", imperative and low-level languages have been the object of a great deal of aggressive compiler work, making the introduction of powerful new optimization technique such as high-level program specialization particularly welcome. Moreover, even for traditional optimizations, high quality analysis of pointer and array computations in such languages can lead substantial (and hitherto only theoretically possible) performance gains by eliminating the need for making grossly conservative approximations to possible side effects of writes [8][13]. The PARTICLE system joins a recent surge of interest in applying high quality but practical partial evaluation and program specialization technology to the realm of low-level and imperative languages.

1.1 The PARTICLE System: A Framework for Experiments in Program Specialization

1.1.1 The General Framework

A primary motivation for constructing the PARTICLE system is not the desire to give the world "a better compiler" or to advance a particular analysis methodology, but to provide a *framework* in which to explore the benefits and costs of simple program specialization mechanisms guided by analysis collected using the technique of abstract interpretation, and to empirically study the availability and character of static information about program quantities. This thesis reports the preliminary results of one experiment within that framework; it is anticipated that future results will investigate variations offering different compile time/run time tradeoffs. Thus, while this thesis refers to the decisions made in the design of PARTICLE, and to the algorithms used within PARTICLE, it is important to keep in mind that the most of the decisions being discussed concern the design of a particular subsystem on top of a general substrate.

1.1.2 The Current Experiment

The current PARTICLE experiment is designed to study the quality of specialization available by collecting data using two extremely powerful but slow mechanisms: The use of the technique of symbolic evaluation to perform highly precise abstract interpretation, and the use of an arbitrarily extensible set of rich abstract domains to capture high quality information about program behavior and data. In accordance with a general philosophy of offering the user a wide array of controls over the quality, character, and speed of the specialization process, PARTICLE permits the user to trade off the precision and compile time cost of PARTICLE's program analysis in a variety of ways.

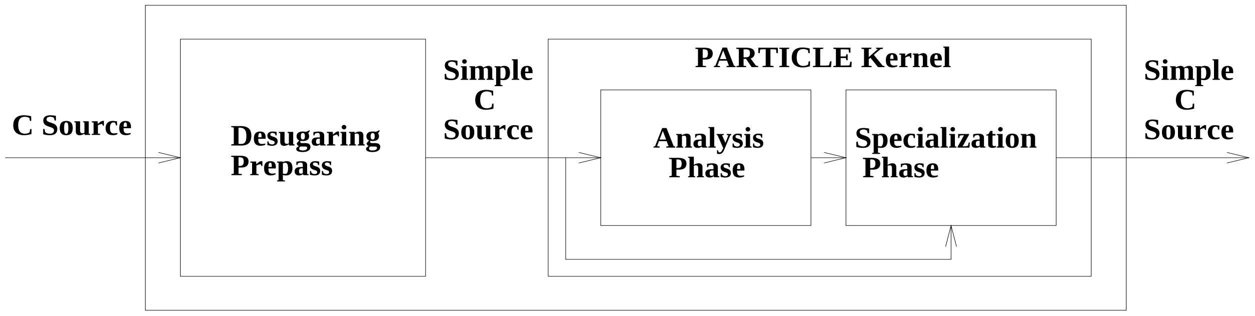

Figure 1-1: The diagram sketches the general structure of the PARTICLE system. The system is divided into two major phases, of which the specialization kernel is by far the most complex. The kernel is itself divided into two major stages communicating by means of a simple and well-defined interface; the analysis phase of the kernel symbolically evaluates the input program in an attempt to discover opportunities for specialization. The specialization phase of kernel operation considers possible patterns of high-level specialization in light of the cost/benefit estimates of the low-level specializations found possible during program analysis user time/space tradeoffs.

As illustrated in figure 1-1, PARTICLE currently operates in two major phases. During the first phase of PARTICLE operation, a program written in the C programming language is desugared into a syntactic and semantically simpler representation of that program. The desugaring process canonicalizes program constructs and types, makes all type conversions and type specifications explicit, and applies a number of other minor transformations to the input program. The desugaring of the input program considerably simplifies the design of the second phase of PARTICLE by allowing that phase to avoid handling all but a small basis set of constructs, and to sidestep the need to correctly deal with complications during the parsing and static semantic processing of the input program. The desugaring phase of PARTICLE is performed by a stand-alone language processor that is conceptually distinct from the remainder of the PARTICLE system, and that plays no role in the specialization of the residual program.

The second and by far the most important phase of PARTICLE operation is the specialization kernel, which is responsible for discovering and performing all of the specializations to be realized by the system. As shown in figure 1-1, the specialization kernel itself exhibits a unique two-phase structure. In the first phase of kernel operation, program analysis is performed to collect information on opportunities for specializations within the input program. The second phase of the kernel uses the information on legal specializations collected during the first phase to create a specialized version of the original program. Delaying specialization decisions until program analysis completes allows the system to expclitly weigh the benefits associated with different possible patterns of high-level specialization in light of global knowledge of program beheavior and user specified time/space tradeoffs. Such a strategy allows PARTICLEto avoid the pitfalls associated with the trivial or "greedy" specialization algorithms commonly used in program specialization systems. The character of each of these phases of the specialization kernel will briefly be described below.

1.1.2.1 Program Analysis through Symbolic Evaluation

During the first phase of the program specialization process, the input program is symbolically evaluated as completely as possible given any available static knowledge of program data (including any input data provided by the user) and user-imposed constraints on running time. During symbolic evaluation, PARTICLE records the estimated time and space savings of any discovered opportunties for low-level (statement and expression level) specialization.

Like many forms of compiler program analysis, the program analysis performed by PARTICLE can be viewed as a form of abstract interpretation. When seen in this light, PARTICLE differs from most abstract interpretation algorithms in making use of an unusually rich and arbitrarily extensible abstract domain, and an analysis procedure that can simulate the program's run time flow of execution as closely as desired given the available level of knowledge about program data and the need to guarantee termination of analysis.

While performing abstract interpretation using the rich abstract domains employed in PARTICLE requires considerably more machinery and compile time than is needed for simpler domains, the information obtained about program data values is correspondingly more complete and precise: The maintainance of high quality approximations to run time data values allows the recognition of invariants of program behavior that are invisible to simpler analysis. In certain cases, highly precise modeling of run time data is important for knowledge about further data to be deduced; in a few cases (particularly involving arrays and pointer manipulation), this "multiplier effect" can be dramatic, and the discovery of *any* knowledge of important characteristics of program behavior hinges on the quality of analysis employed for some subset of program data [8].

Similarly, the use of full-bodied symbolic interpretation can impose significant compile time overhead, but is capable of performing extremely high quality analysis, an advantage that is particularly pronounced in the analysis of function calls and program loops: As discussed above, virtually all existing compiler systems make little attempt to exploit the large amounts of static data available at the interfaces between functions; when coupled with specialization at the function-level, such information can be used to perform powerful interprocedural optimizations. The opportunities for optimization in program loops are less dramatic but still important: Despite the fact that most programs spent the vast majority of their run time within a small fraction of their code [33], the program analysis performed by existing compiler systems does not reflect this imbalance: The quality of the static analysis performed on program loops and recursive functions is likely to be substantially *less* precise than the analysis performed on other portions of the program. In many circumstances, an analysis strategy that that better reflects the distribution of run time activity within a program can capture substantially more knowledge about program quantities and behavior, thereby permitting the application of a larger set of optimizations. In particular, the ability to precisely simulate the run time flow of execution within a program represents an essential prerequisite for the capacity to shift arbitrary classes of computation from run time to compile time and for tightly bounding the possible behavior of program loops. While supporting the use of arbitrarily precise program control flow analysis, PARTICLE guarantees that such analysis will terminate within some finite time regardless of the run time behavior of the program being simulated. The user is permitted to vary the threshold at which fine-grained control flow simulation of program loops gives way to more traditional and coarse fixed-point iteration.

The opportunities for low-level specialization noted during symbolic evaluation contain the crucial information necessary to allow a selection of a maximally beneficial pattern of high-level specialization during the second phase of PARTICLE; the high quality and flexible character of analysis employed by PARTICLE can allow the discovery of possible optimizations far beyond the reach of existing compilers.

1.1.2.2 The Specialization Phase

When program analysis terminates, the second phase of the program specialization process begins. For each function PARTICLE selects a set of specializations of that function that are judged to be the most beneficial and least costly and creates a specialized program that makes use of these function-level specializations. A central theme of this process is the creation of specialized functions to take advantage of similar patterns of usage permitted by different sets of call sites. In order to make a high quality specialization decision for a given function, PARTICLE explicitly balances the performance benefits associated with the creation of specialized functions highly tuned for particular call sites against the space cost associated with the creation of such functions, and attempts to come up with a solution that maximizes the performance benefits of specialization while minimizing the associated space costs. In accordance with PARTICLE's general philosophy of providing the user with control over the character of the specialization process, PARTICLE permits the user to specify the relative weighting of space and time costs and benefits, thereby alternatively directing PARTICLE to seek some perceived optimum between allowing the use of function specializations only when they are likely to yield extremely high performance boosts, and attempts to maximally exploit any possible performance gains from specialization even when they result in the proliferation of trivially different specialized functions.

While the specializations performed by PARTICLE have considerable overlap with those performed by traditional compilers and in some cases go far beyond them, PARTICLE is designed to *complement* such systems, rather than to replace them. Rather than transforming input code to object code, PARTICLE operates as a source-to-source transformation system. This policy allows particle to delegate traditional but demanding optimizations to an traditional aggressive post-processing compiler and allows PARTICLE to concentrate on implementing a few optimizations in a high quality manner without compromising other aspects of code quality or portability.

1.2 Shortcomings of PARTICLE

1.2.1 Introduction

Earlier sections have sketched the motivations for PARTICLE's approach and characterized some of the benefits arising from that approach. For balance, this section turns to briefly examine some of the ways in which the current PARTICLE system has failed to live up to its promise. Some of these shortcomings arise from high-level decisions made in the design of that experiment, while others result merely from lack of time to completely implement an intended design. The most serious failings in each category are discussed below.

1.2.2 Unacceptable Compile-Time Resource Consumption

The current PARTICLE experiment requires enormous amounts of time and space to operate. The extravagant resource demands of the current system preclude its application to programs of large or even moderate size – the very software systems in which the tension between good softare engineering and efficieincy is the greatest. Even for small programs, the excessive resource consumption of PARTICLE seems likely to make its application undesireable to all but a small fraction of input programs: While the quality of PARTICLE's analysis and specialization mechanisms far exceeds that of traditional compilers, the times required for analysis by PARTICLE are many orders of magnitude beyond those needed within more conventional systems. Processing by PARTICLE is therefore likely to be beneficial only for programs offering substantial opportunity for specialization and whose expected total run time is very large.

There are many factors contributing to PARTICLE's extravagant resource consumption, but many of them stem from *architectural* aspects of the current analysis system rather than from simple cosmetic shortcomings within its implementation. In particular, while PARTICLE's analysis framework is highly extensible, certain aspects of that analysis are less flexible and cannot be easily "scaled back" by a user seeking faster analysis. Two of the most important of suchfixed costs arise from PARTICLE's failure to offer any means for throttling the simulation of function calls , and the system's rigid maintainance of an extravagantly fine-grained model of program state throughout the symbolic evaluation of a program. The inflexibility of each of these aspects of program analysis implies that even a minimal-cost analysis performed by the system will tend to be fairly expensive.

The analysis performed by PARTICLE currently consumes far more compile time resources than are likely to be justified by the run time performance increase offered by the system: The specialization of even small programs (such as those discussed within chapter 6.) not infrequently demands hours of computation and many megabytes of memory. The compile time/run time tradeoffs offered by the current system are unlikely to be acceptable for all but a small fraction of realistic programs, and confine effective experimentation with the system to work with a small subset of realistic programs.

1.2.3 Omissions in Modeling

While PARTICLE handles a fairly rich set of input programs, time constraints have left many serious lacunae within PARTICLE's modeling of the C language and the popular C language libraries. Chapter 5 discusses a number of extensions to PARTICLE that seek to remedy shortcomings and omissions within the modeling capacity of the current system, but it is worth mentioning some of the most important of these failings here.

- **Failure to Simulate the Heap Libraries** PARTICLE is currently incapable of modeling the behavior of heap managment routines associated with the standard C library. Such routines are used within virtually all large-scale C programs, and are used to create the heap data structures that typically store the vast majority of the data in such programs. As will be discussed in section 5.2.3, low-level simulation of such routines is possible in theory, but virtually useless in practice. As a result, the lack of special-case capacity for modeling the operation and behavior of heap memory management renders PARTICLE unable to perform even coarse analysis of the heap structure or activity of programs being processed, limits the application of PARTICLE's high quality analysis to the stack and global data areas, and forces the system to make grossly conservative assumptions about the possible possible effects of writes to pointers within the heap.

- **Failure to Model Aggregates of Unknown Size** Although some computer languages require that the size of all data aggregates within a program be statically constant, C imposes no such restriction – data within any of the three major segments can be of unknown static extent. As willl be seen in section 5.2.2, the modeling of regions of imprecisely known length requires some care, and PARTICLE currently has no capacity for analyzing programs manipulating aggregates of unknown extent. This restriction enormously restricts the set of input programs to which PARTICLE can be legally applied.

- **Small Lacunae** There are a number of features of the C language for which PARTICLE currently offers limited or no support. Such shortcomings virtually always stem from pragmatic implementation tradeoffs (imposed by a lack of sufficient implementation time) rather than from any conceptual- or design- level difficulties in formulating a more complete approach. At the same time, such shortcomings are notable for contributing a number of important in PARTICLE's support for the C language and in further belying the system's claim to handle completely general input programs. A few of the more important failings in this class include the incompleteness of implementations for simulating program execution in the context of fully general patterns of control flow (see section 4.3.4.), PARTICLE's lack of support for the C language "union" construct, the inability of the system to handle programs divided into multiple source files, and the failure of PARTICLE to provide even trivial support for calls through incompletely known pointers to functions. While such shortcmings can be easily addressed within future versions of PARTICLE, it is important to stress that in a number of ways the current system fails to entirely live up to its goal of handling all features of the C language.

1.2.4 Incomplete Specialization Strategies

The specialization strategy advanced by PARTICLE sets the system apart from previous program specialization projects. In particular, PARTICLE's division of the specialization process into separate analysis and committment phases provides the system with the necessary data to perform specializations by explicitly weighing the benefits associated with different patterns of high-level specialization in light of global information about program behavior and user-specified time/space tradeoffs. The design of PARTICLE thus offers the potential for employing considerably more sophisticated specialization strategies than were used within previous projects. However, while PARTICLE currently collects information necessary for performing an extremely high quality and sophisticated choice among specialization strategies, the implementation of the selection mechanism whose design is detailed in section 4.4.4 is not yet implemented. As a result, the system at the time of writing makes use of a far simpler and naive specialization strategy, and all of the preliminary performance results offered in Chapter 6 were made using that system Although the implementation of the more sophisticated specialization strategy should be completed shortly, it is not offered by the system. Thus, while PARTICLE can make a claim to offering a framework that *permits* specialization decisions to be made given far richer information than was available within previous projects, the mechanisms needed to take full advantage of that information are not yet in existence.

1.2.5 Conclusions

This section has characterized the primary ways in which the current PARTICLE experiment has failed to live up to its goal of providing a practical and extensible specialization system for low-level imperative languages. It is hoped that by discussing such failings at the beginning of this document the reader will benefit from a better understanding of both the strengths and weaknesses of the current system, and the extent to which the system designed sketched in later chapters is realized in practice. Chapter 5 discusses promising means for remedying many of these failings in later version of PARTICLE.

1.3 Overview of Document

This thesis presents an overview of the PARTICLE program specialization system, emphasizing the motivations for its approach, the system design and implementation, and potential avenues for future work on that system. It should again be stressed that PARTICLE is still very much in the process of evolving; the current document should not be taken as a defense of a certain strategy as optimal, but as an attempt to set out what has been learned thus far through a very limited exploration of a large parameter space. Chapter 2 briefly reviews work related to program specialization in the areas of compiler design and partial evaluation. Chapter 3 examines the some of the motivations for program specialization, provides a brief tutorial on the character of static knowledge and its interaction with specialization, and discusses a number of general decisions made in the design of PARTICLE. Chapter 4 turns to examine *how* PARTICLE works, and briefly sketches the principle abstractions and algorithms used within the system with an eye towards explaining the overall system choreography. Chapter 5 offers preliminary speedup measurements for a number of pieces of code specialized by the system, and tries to provide some insight as to the origin and implications of these performance gains. Chapter 7 sketches a number of potentially fruitful avenues for future research that would address shortcomings of the current system or further enhance its functionality. Finally, chapter 10 offers a concluding summary of the successes and shortcomings of the research and suggests immediate priorities for future work within PARTICLE.

Chapter 2

Related Work in Program Specialization

2.1 Introduction

This chapter surveys past work in areas relevant to the design of PARTICLE with particular emphasis on the manners in which PARTICLE is similar to and different from past systems. Although the details of the current PARTICLE system are complex, the conceptual basis for this system is rather simple and emphasizes two major approaches to the problem of high quality compilation:

- The use of multi-level *program specialization* to optimize a source program to particular patterns of data that will obtain at run time.
- The maintainance of arbitrarily rich representations of knowledge about run time data values and the use of full-bodied analysis strategies in order to capture maximal information about patterns of run time data and control flow for possible use in program specialization.

There are two major collections of work that share aspects of PARTICLE's charter: Work studying the program specialization technique of *partial evaluation*, and past efforts employing symbolic and abstract interpretation for program analysis and optimization. There is a large amount of variance in emphasis and goals within the each of these domains, and the discussion below concentrates on aspects of each area which are most relevant to the design or philosophy of PARTICLE.

2.2 Partial Evaluation

The program specialization technique of *partial evaluation* has been the subject of an increasingly broad range of interest within the past decade. Although first described in 1971 by Futamura as a means of compiler generation [20], during both the 1970s and within the last 5 years a number of studies have studied its use in program optimization. Before discussing the particulars of this work, I will provide a brief conceptual overview of partial evaluation.

Partial evaluation is a form of program *transformation* which takes as input a program and some subset of the inputs for that program. The partial evaluator outputs a *residual* program having the special property that when it is supplied with the *remaining* program inputs (i.e., those not supplied to the partial evaluator), it will execute in exactly the same manner as the original program would have executed had the original program been given *all of the inputs together*. Slightly more formally, a partial evaluator $\mathcal{PE}$ is defined by the relation

$\mathcal{PE}(p(a,b),x) \Rightarrow p'$

where

$\mathcal{I}(p',y) = \mathcal{I}(p,(x,y))$

where $\mathcal{I}$ represents an interpreter for the source language of the program p, a and b represent formal input parameters to p, and x and y represent actual input parameters to p (i.e., the input to the program). p' is the residual (specialized) program output by $\mathcal{PE}$, and the '=' should be taken as a behavioral equivalence relation. It should be noted that for clarity I have assumed that p takes only 2 arguments and that the partial evaluation is only performed with respect to the first input. In fact, p may accept any number of inputs, and any subset of the p's inputs may be specified as fixed. Intuitively, then, partial evaluation is a means of taking a program and some fixed and known input data and generating a version of that program *specialized* to that input data [18].

Although partial evaluation is defined as the process of specializing with respect to input data, it is important to note that the mechanisms in a partial evaluator that permit specialization with respect to input data typically also permit the specialization of a program with respect to any static knowledge of program data that may be extracted from the program text. Thus, the knowledge of program quantities exploited by partial evaluation typically derives from two sources: A partial specification of program input, and any information about run time values that may be discovered from analysis of the source code for the program itself. For reasons that will be discussed in section 3.1.3.5, it is frequently the case that an incremental gain of knowledge about a program's inputs will allow greater potential for new specializations than will an incremental gain of knowledge about an arbitrary internal program value. However, the potential for specialization on the basis of information contained he program source code alone can still be enormous.

While the definitions given above leave open the question as to how the specialization with respect to the statically known data is accomplished, partial evaluators typically try to take maximal advantage of data available statically. In particular, while the definitions above do not require it, specialization generally involves an attempt to perform as much of the program's computation as possible at compile time, given the available knowledge about program input. Thus, partially specified program input is used to shift as many computations as possible from run time to compile time, thereby improving run time performance.

2.2.1 Taxonomy of Partial Evaluators

Existing partial evaluators typically fall into one of two categories: *online* and *offline*. Each of these classes is associated with differing goals for program specialization and means of program analysis. While PARTICLE shares more affinity with highly optimizing online partial evaluators rather than with offline evaluators, a brief discussion of both of the two major types of partial evaluators is included below.

2.2.1.1 Offline Evaluators

The Futamura projections[20] describe a means of automatically producing compilers and compiler generators for a language L by recursive applications of a partial evaluator to itself and an/or an interpreter for L. Research in this area demonstrated that such self-applicability by partial evaluators is practical only in systems which had very simple program analysis and specialization kernels) [7][56]. This realization has led to an increasingly sophisticated class of *offline* partial evaluators which allow self-applicability at the cost of a less powerful capacity for program analysis and specialization.

The defining feature of the offline method is the decomposition of the program specialization process into two phases: A stage known as "binding time analysis" (or BTA) which analyses the program text to discover which program quantities will be statically known at compile time and annotates program constructs to indicate whether they will be able to be evaluated (or "reduced") at compile time. A second stage then symbolically evaluates reducible portions of the program to effect the specializations that were noted as possible in the first stage. The character of these two stages is sketched below.

During BTA, the staging behavior of the input program is analyzed and constructs are annotated with information indicating whether they should be reduced during the second phase or left residual in the program until run time. The program analysis abstracts away program quantities from their concrete values into some small finite domain, and is performed under the conservative assumption that all paths of execution are possible at run time. In the simplest offline systems, the domain used during BTA is limited to expressions of the stage (run time vs. compile time) at which an entire value will be known [41]. More complex but higher-quality schemes enrich the domain to capture finer-grained information about partially static structured data (so that certain *pieces* of structured data can be recognized as known or unknown, rather than forcing the entire composite value to be characterized as one or the other [41][9] [37]. Information about the binding time of values is then used to determine which constructs will be able to be reduced during symbolic evaluation (when the static information about known values will actually be known). It is important to stress that while the BTA performed during offline evaluators maintains information about the *binding time* of pieces of information, it does not collect or make use of information concerning any *concrete values* that may be associated with program quantities or expressions. Binding time analysis merely attempts to characterize the range of data that *will* be precisely known during the second stage of offline specialization (symbolic evaluation), but does not itself collect, deduce or manipulate this data. Similarly, while BTA makes reduce/residualize *decisions* for program constructs on the basis of this information indicating the binding time of the values returned by these constructs, the actual *performance* of these reduction decisions does not take place until the second phase of operation, when symbolic evaluation used to compute that data deduced as available during BTA.

In the second phase (the "specialization" phase) of offline partial evaluation, the reductions specified by the first phase are performed. During this stage, the system symbolically evaluates the user program on available static information (provided either by specifications of user input or available from the program text), and evaluates all constructs associated with "reduce" annotations by the first phase. Because the second phase of program specialization is needed only to effect reductions on statically known data (as well as to residualize any recursive calls), the offline evaluator needs to maintain information only on those data values that are exactly known. As a result, the overhead associated with the symbolic evaluation tends to be rather minor (involving little beyond the mechanisms needed for traditional program interpretation), and the specialization phase can be highly efficient.

By decoupling the reduce/residualize decisions from symbolic evaluation, offline evaluation can be performed efficiently and simply. In particular, BTA makes use of small abstract domains and can be rapidly performed using standard analysis techniques, and the interpretive overhead associated with symbolic evaluation is minimized by avoiding the need to represent incompletely known values and by evaluating only those constructs known to be reducible. Unfortunately, such simple analysis also forces the reduce/residualize decisions to be made in the absence of high quality information concerning program values. As a result, BTA can incorrectly classify a value that would be statically determinable during symbolic evaluation as unknown. This misclassification of a value can lead both to the direct failure to to discover certain legal specializations and to the compounding of the problem by misclassifying additional values that depend on this value in the same manner.

In [47], Weise characterizes two major sources of imprecision in offline analysis: The inability of BTA to use information about concrete values of program quantities in making reduction decisions, and the failure of the offline framework to support effective generalization to capture the commonalities between different run time execution contexts. Each of these shortcomings is briefly described below:

2.2.1.1.1 Specializations Based on Run Time Values Binding time analysis associates reduce/residualize annotations with each construct in the input program. In order to be safe, such annotations must be valid across all of the possible run time execution contexts compatible with the information collected during BTA. Because BTA only collects information regarding the binding time of program values, and has no access to information concerning the particular run time values associated with static quantities, this restriction can greatly constrain the subset of legal specializations discovered by an offline evaluator. In particular, there are an important class of program specializations which can only be recognized and exploited given information on the actual run time values associated with program quantities. In such cases, knowledge of the binding time of the program quantities is insufficient (and in some cases, irrelevant) to know whether a given optimization can be performed – particular ranges of run time values may permit a specialization, while other values will rule it out, and because BTA lacks any information concerning the concrete values that will be associated with program quantities at run time, it is forced to conservatively assume that the specialization is not possible. In many such cases, lack of knowledge about the particular run time values associated with certain program quantities can also prevent the acquisition of accurate binding time knowledge concerning other quantities. A few examples of these phenomena are shown below.

At the expression level, there are a range of cases in which information concerning the binding time of subexpressions is irrelevant to the binding time or value of the expression as a whole. For example, while the exact value of a certain program quantity v may not be known, it may be statically determinable by symbolic evaluation that it is greater than zero. In the presence of such information about the range of possible values of v, it is determinable that the predicate $v > 0$ is true.[1] During BTA, however, such information bounding the value of

[1]Of course, recognizing the truth of this predicate even in the presence of knowledge concerning run time values requires the maintainance of fairly sophisticated abstract domains.

v is not available, and all that can be deduced is that v is not statically known (not exactly known). Since there exists *some* cases in which a value of v that is not statically known would yield a result to the $v > 0$ predicate whose truth value is unknown, BTA must conservatively assume that the value of v that would be manipulated during symbolic evaluation could be such a value, and must record the expression as residual (not capable of being reduced). As a result, not only will the predicate unnecessarily appear in the residual program, but BTA will also be forced to conservatively residualize any constructs that make use of the result of the predicate (such as an enclosing *if* statement).

Another set of cases in which actual knowledge of concrete values are needed to discover expression-level specializations involve optimizations based on algebraic properties. For example, it may be deducible during symbolic evaluation that v_1 is known to be zero. Unfortunately, during BTA the system is limited to knowing that v_1 will be statically known during symbolic evaluation. As a result, partially reducible expressions such as $v_1 + v_2$ (v_2 unknown) will not be discovered as capable of being reduced (in this case, to v_2), and will be marked as residual. Worse yet, expressions such as $v_1 * v_2$ that are entirely reducible to constants will not only be recorded as residual, but the result of this expression will also be viewed as "not statically known" and will thus preventing the discovery of further static data and possibilities for specialization.

Two of the previous examples illustrated cases in which lack of knowledge during BTA of the exact value of one program quantity not only prevented the recognition of of a valid specialization but also led to the failure to recognize another value as statically known. A similar but far more common and serious case occurring at the statement level is illustrated in figure 2-1. Since BTA operates with no knowledge of the run time values associated with program quantities and does not actually *evaluate expressions* using such values until the specialization phase, the actual truth value returned by predicates of *if* statements will remain unknown during BTA. As a result, even if the binding time analysis is aware that the predicate of the *if will be known during symbolic evaluation*, it cannot make use of the truth value associated with the predicate to guide the binding time analysis of this construct and will be unable to determine which branch of the *if* will be taken at run time. Binding time analysis will thus be forced to make the conservative assumption that *either* branch could be executed at run time, and will associate "unknown" values with a program quantity assigned an "unknown" value in *either* the conditional or alternative of the *if*.[2] Thus, the general inability of offline evaluators to determine which branch is taken during BTA *even when it is statically determinable during symbolic evaluation* can substantially dilute the binding time information collected on other program quantities assigned to within the branches. Since *if* statements are common in code, it is to be expected that inability to perform accurate binding time analysis on such statements may have a significant negative impact on the quality of program analysis.

As a final example of the inability of BTA to discover important specializations, consider a *while* loop in which simple static analysis is able to demonstrate that the loop will only be iterated 8 times and can thus be completely unrolled and recursively specialized. Unfortunately, since information concerning the values of program quantities is not available during BTA, BTA will be unable to discover the proper amount of loop unrolling at the point where specialization

[2]Note that if a program quantity is assigned different known values in each of the different branches of the *if* statement, the quantity should still be marked as "known" if the predicate is also known.

```
x = 1;          /* suppose x will be statically known */

/*
BTA knows that during symbolic evaluation the value of this
predicate will be statically known, but BTA does not itself have
access to this value, and thus cannot know which side of the
branch will be taken
 */

if (x > 0)
   a = 1;       /*  if ``then'' branch taken, a statically known */
else
   a = b;       /*  if ``else'' branch taken, a unknown */

/* during BTA, unknown if a will be known or unknown here
   during symbolic evaluation */
```

Figure 2-1: Because concrete data values are never available during BTA, it is not always possible to recognize when static data is available. In this example, during BTA it is not possible to know which direction the branch will take. As a result, BTA must make the conservative assumption that x will not be statically known at the end of the conditional

decisions are made and will lose the ability to unroll and specialize the loop.

Although techniques involving polyvariant binding time analysis and continuation-passing-style conversions can address some of the difficulties presented above for functional languages [10][41], they can lead to substantial code duplication and the introduction of additional complications in program to analysis [47].

The compile time efficiency of offline specializers benefits substantially by restricting symbolic evaluation to those expressions that are denoted as reducible. Unfortunately, because reduction decisions are (conservatively) made in the absence of the information concerning program values that is available during symbolic evaluation, there are several cases in which expressions which *could* be fully evaluated during static evaluation will be conservatively annotated as residual. As a result, the symbolic evaluation during specialization will leave these reductions residual, despite the fact that it possesses enough information to exactly evaluate such expressions. Thus, the poverty of information concerning program values available during BTA allows program analysis and specialization to terminate rapidly, but also conspires to provide a cap on the specializations performed during symbolic evaluation. As a result, the specializations decided upon by BTA may prove a weak subset of those realizable with more complete information on program values.

2.2.1.1.2 Generalization of Program Information An additional difficulty with offline evaluation noted by Weise in [47] concerns the inability of BTA to recognize and exploit *commonalities* between different contexts in which a program construct may be evaluated. In particular, specializations of a piece of code may only be legal under certain sets of conditions (e.g., when values are associated with particular concrete values). In order to share a piece of code among several different execution contexts (e.g., to share a specialized function among several different call sites, or a loop body specialization among several different iterations), the program specializer must be able to recognize that precisely the same reductions are legal for each of the possible contexts which share the use of the specialized code. Aside from exceptional cases (e.g., the appearance of an expression such as $v * 0$, where an identical reduction can be performed regardless of context), the applicability of a particular reduction typically depends on specific features of values in the context of execution (e.g., the fact that a program quantity is associated with a particular precise value, or with some range of values). Thus, the *recognition* of the contexts under which particular specializations are legal requires precise knowledge of program values. Such information is not available during BTA, and cannot be used to make reduce/residualize distinctions. This inability to recognize similarities in execution context can lead to excessive duplication of code and trivial overspecialization in attempts to guarantee that the possible specializations associated with each particular context are exploited. (For example, some offline approaches attempt to duplicate code until each copy of the original code is associated with only a single run time context [10][41].)

While the widespread duplication of code to get around an inability to recognize commonality in execution contexts is clearly undesirable, the alternative is to make the drastically conservative assumption that the contexts share no potential for common specializations, and to risk a failure to exploit substantial commonalities among the set of different contexts possible for a construct. As was the case above, because of the (time-saving) restriction of the use of symbolic evaluation to cases where the expressions have been annotated by BTA as reducible, the failure to effectively discover the full set of reductions possible at a program point can seriously limit the quality of static analysis performed during symbolic evaluation.

As was the case with the difficulties introduced by incomplete context information, offline evaluation strategies have been proposed that attempt to deal with lack of effective generalization ability during BTA. Unfortunately, such techniques tend to complicate the specializer operation, frequently duplicate or excessively transform code, are sometimes not of general applicability, can prove problematic or unworkable in the presence of recursion, and can significantly complicate guarantees of specializer termination (many offline evaluators require explicit user-created annotations to guarantee convergence of generalization during the specialization phase) [47]. In general, the creation of a single specialized function usable from many different call sites (possibly including sites inside the body of function) is greatly complicated by the inability to generalize program contexts in a straightforward manner at the point where reduce/residualize decisions are made.

2.2.1.1.3 Offline Evaluation: Summary The discussion above has provided a brief sketch of the character of offline partial evaluation. Offline evaluation offers rapid specialization and realistic and effective self-applicability. These characteristics alone make it a powerful tool for program transformation. Unfortunately, the very partitioning of the specialization problem that yields such important advantages can also hamper or prohibit the exploitation of important classes of program specializations, and can significantly complicate a relatively simple overall strategy. While this thesis borrows relatively little from the practice of offline evaluation, research in offline evaluation remains an area of great promise and dynamism; the technique has advanced greatly since its inception, and has significant potential both for practical application on its own and for influencing other compilation techniques (particularly the practice of on-line evaluation: see next section) [22][47]. Moreover, within the past several years, important progress has been made towards applying off-line partial evaluation to imperative and low-level languages (work that has included the construction of an offline partial evaluator for C)[51]. Despite their shortcomings, offline techniques represent a promising area of growth in the spectrum of compiler analysis techniques.

2.2.1.2 Online Evaluators

2.2.1.2.1 Introduction *Online* partial evaluators sacrifice the rapid running time and self-applicability of offline evaluators in order to gain access to a much richer knowledge of program data values when making specialization decisions [56]. While offline evaluators separate the partial evaluation process into a binding time analysis phase which annotates those constructs to be reduced and a symbolic evaluation phase that performs the calculations necessary to perform these reductions, online evaluators discover possible specializations during a full-bodied symbolic evaluation of the entire program.

During the process of symbolic evaluation, operations in the program are performed as precisely as possible given the incomplete information available to the symbolic evaluator. For example, if the online partial evaluator ever has enough information about program values to allow it to evaluate a construct (whether a primitive operator, a control-flow construct, a library function call or a call to a user-defined function), the partial evaluator will do so. Where exact knowledge of program values is not possible, *approximations* to the such values are used, and operations are simulated on such values just as they are simulated on precisely known values. Where exact patterns of control flow are not known, the symbolic evaluator approximates the effects of all possible control flow patterns. Even in the case of program loops and function calls, symbolic evaluation will directly execute the constructs wherever possible, safe, and desirable[3]. Such precise simulation can permit markedly better analysis of program behavior than is possible under traditional compiler analysis, and can allow the discovery and application of important specializations invisible to less exact analysis (see section 4.3.3.2.2.3.). Unfortunately, it can also be extraordinarily expensive, and when applied naively can lead to compiler running time that as long or longer than that of the program being evaluated. While in certain situations such long running times for symbolic evaluation can be effective in shifting large proportions of the computation from run time to compile time (e.g., the effects of a loop with known bounds doing expensive computations on statically known values can be computed exactly, and a deep recursive function call calculating some result from statically known values can be collapsed to a single value), in other cases such high-precision simulation offers little or no gain in performance at run time, at enormous compile time cost (for example, the precise simulation of a huge loop with known bounds whose body is performing computation on values

[3]Note that PARTICLE allows the user to impose limitations on the full-bodied simulation of loops and recursive function calls at two different granularities, in order to permit more strict timing constraints to be placed on symbolic evaluation.

that are completely unknown at compile time will yield very little static information). In contrast, offline evaluators may miss substantial opportunities for specialization because of the ignorance of data values during BTA, but also avoid much of the overhead often associated with the full-bodied simulation of code where it is not fruitful to do so. Figure 2-2 illustrates one example where online evaluation may spend a considerable time simulating program behavior to little effect, while symbolic evaluation during offline evaluation will avoid such costs. (In fairness to online evaluators, it should be noted that while there are frequent cases in which such systems will simulate fine-grained execution without computing any further static information about program quantities, in some instances the very precision of this simulation allows the discovery of opportunities for other specializations. For instance, such computations often still permit opportunities for high quality loop unrolling and the collapsing of conditionals (e.g., the loop in figure Figure 2-2 could below could be unrolled with the execution of each iteration).)

```
n = 10;
x = u;                  /*  u unknown */
a = 1.0;

while (n > 0)           /* value of this always known */
        {
        a = a * x;      /* after first iteration, a unknown */
        n = n - 1;
        }
```

Figure 2-2: This example illustrates a case in which the fine-grained simulation possible in online evaluation has the potential for considerably increasing compile time without any benefit to static analysis. While the loop is simulated exactly, all computations take place on unknown values, and no new static information is computed.

While offline evaluators make specialization decisions in a binding time analysis phase executed *prior* to symbolic evaluation, online partial evaluators have generally taken the approach of actually performing program specialization during symbolic evaluation wherever static knowledge permitted it. In such frameworks, wherever exactly known values are available or operations could be executed completely, their results are incorporated directly into the residual program in place of the code expressing the original operations; in cases where insufficient information was available to allow the partial evaluator to reduce an operator or simulate a construct directly, the execution of construct was delayed until run time, and the construct itself was output by the partial evaluator as part of the residual program [56][25][49][4]. (See, for

[4]In such systems, there may be some important variations on this process necessary in order to handle the

example [56].) On account of the widespread (and perhaps universal) adherence to this practice in online evaluators, it should be stressed that even in online evaluators the symbolic evaluation process and the job of making specialization decisions are conceptually distinct, and need not be performed simultaneously. In particular, the distinguishing feature of online evaluators lies in the *use* of information gained during symbolic evaluation in the discovery of specializations, in contrast to the offline practice of committing to a set of specializations prior to a vastly scaled-down form of symbolic evaluation. While existing online evaluators may choose to "greedily" *commit* to performing discovered specializations during symbolic evaluation itself, such a practice is not strictly necessary: the information collected concerning legal specializations discovered during symbolic evaluation can also be saved away and analyzed once symbolic evaluation has completed and complete data concerning program behavior is available. This is the approach adopted by PARTICLE, as will be discussed in chapter 3.

Because symbolic evaluation in online evaluators requires the modeling of *all* program values and the attempted reduction of all program constructs reachable at run time (rather than the simulation of just those values known precisely and reductions of just those constructs permitting specialization), online evaluation typically requires far more compile time and space than offline methods. At the same time, because online methods attempt to recognize specializations *during full-bodied symbolic evaluation of the program*, online specializers often have access to far greater knowledge of program values and behavior than is possible during either the binding time analysis phase or the limited symbolic evaluation phase of offline evaluation. While offline evaluators can afford to only represent exactly known values during the reduction of statically known constructs in the specialization phase, online evaluators must make use of abstract value domains that capture information about unknown and incompletely known values, and must manipulate this information in a generally and effective manner during symbolic evaluation.

The complete modeling of program state (rather than just the known values) permits the construction of high-fidelity approximations to program state during symbolic evaluation and allows effective characterization of different program execution contexts associated with a given piece of code and the commonalities between them. The availability of such information during program specialization permits online specializers to straightforwardly construct specialized code reusable in a a range of execution contexts, allows easy guarantees of program termination without the need for user-provided annotations or extensive program analysis, and permits the discovery of data-dependent optimizations whose exploitation is difficult (if not impossible) under offline methods. On the other hand, the need to manipulate extended abstract domains and to recognize specializations during symbolic evaluation requires a considerably more extensive and sophisticated specialization kernel than is necessary for offline strategies, and is associated with significant compile time overhead. Moreover, the casting of partial evaluation within the framework of symbolic execution and the size and complexity of the associated of online specialization engines makes effective self-application difficult if not completely unrealistic. As a result, while online evaluators are capable of higher-quality program analysis and specialization than offline evaluators, they can exhibit long running times and are typically incapable of taking advantage of the Futamura projections through self-application.

capacity for context generalization or to allow sharing of existing specializations, but the general mechanism remains the same.

2.2.1.2.2 Past Work In Online Evaluation Although the mechanisms required for online evaluation tend to be substantially more complex than those needed for offline methods, the conceptual character of online evaluation is somewhat simpler than that of offline methods, consisting merely of a symbolic evaluation mechanism coupled with some machinery for use in tasks such as generalizing, guaranteeing termination, and performing program specializations. Moreover, because online evaluation is capable of more thorough program analysis and specialization than offline methods, online specializers have attracted the interest of researchers interested in aggressive optimization technology. In the past two decades, several online evaluation systems have been created, spanning a wide range in terms of the use of manual vs. automatic control, practicality, and the richness of abstract domains employed.

The partial evaluation work of Haraldsson [25] made use of a rich set of abstract domains that allowed expression of knowledge about program values that were either completely known, completely unknown, members of some possible set of values, or known *not* to be equivalent to some value. REDFUN-2's *q-tuples* are used during symbolic evaluation to capture both the value of an expression and the associated residual code, thereby simultaneously communicating of the return value and specialization of a program program expression. Haraldsson's symbolic evaluation framework is unusually sophisticated (e.g., it is capable of making use of knowledge about the truth or falsehood of a conditional's predicate when processing its consequent and alternative), and his specializer is capable of modeling program side effects. Unfortunately, REDFUN-2 is limited to the manipulation of scalar values and is incapable of capturing information about the subparts of compound values. In addition, the specializer is not capable of approximating the return values of recursive functions, and requires user guidance and manual intervention in the specialization process. The REDFUN-2 system could not provide guarantees of specialization termination even for programs guaranteed to terminate at run time, and was prone to unnecessary and undesirable duplication of code.

Schooler's IBL symbolic evaluation system [49] makes use of *partial values* similar in spirit to *q-tuples* in that they capture both value information and the residual code associated with expressions, but are considerably weaker in lacking the ability to express the fact that a value is an element of a set of known values or knowledge that a value is *not* a particular known value. IBL is designed as a supplement to a compiler rather than as a standalone program specializer, and allows no user specification of program values or specialization of the program with respect to input. While IBL can approximate the values of recursive function calls and represent higher-order functions, it has several shortcomings. IBL has a simple strategy that incorporates the code associated with partial values directly into the residual program text; unfortunately this results in substantial code duplication and can lead to the proliferation of common subexpressions. In addition, the IBL system is incapable of approximating the effects *if* expressions or function calls left residual, eliminating two substantial sources of static information during symbolic evaluation. More seriously yet, IBL lacks any ability to guarantee automatic termination of program specialization and relies on user guidance for termination.

The systems of Kahn [30] and Guzowksi [23] fall prey to many of the same difficulties as REDFUN-2 and IBL. Kahn's specializer was incapable of handling any form of side-effect, while Guzowski's system offered only a very weak modeling of side effects. Guzowski's symbolic evaluator cannot approximate the return values of recursive function calls, while the systems of both Kahn and Guzowksi lead to code duplication, are incapable of allowing symbolic inputs by the user, and cannot provide strong guarantees of specializer termination.

By far the most practical, high quality and complete online specialization system yet developed is the FUSE system designed Weise [56], which grew out of program specialization work by Berlin and Weise [6][5]. FUSE makes use of code/value pairs called "symbolic values" similar to the partial values of IBL and the q-tuples of REDFUN-2. Although it lacks the side-effect handling, conditional contexts, and rich abstract domains of REDFUN-2, FUSE is unique in providing a considerably more valuable set of characteristics: strong guarantees of automatic specializer termination for all input programs which terminate at run time, a specialization strategy that provides for effective reuse of function specializations, a code generation strategy that prevents the duplication of code that plagues other online evaluators, and the ability to exploit user specifications of program values. FUSE operates on input programs written in the language *Scheme*. The evaluator is designed to specialize the functional subset of Scheme, although very limited support is available for the handling of side-effects. FUSE accepts user specification of input specifications, and evaluates the input program as much as possible given the available static data. During evaluation, FUSE use graphs as an intermediate representation in order to avoid the code duplication problems often encountered during partial evaluation, to express partially known type information, and to allow greater flexibility in making the reduce/residualize discision; a code generation phase following evaluation is used to map the dataflow graphs into Scheme code [55].

Symbolic values in FUSE can represent scalars, structured objects, and higher-order functions, and may express values that are completely known, completely unknown, or only partially known (e.g., cases where the general atomic type of the object is known, but where the value is unknown, such as "a natural"). In addition, FUSE is capable of representing compound values (including only partially known compound values, such as a cons cell whose *car* is known, but where the *cdr* is unknown) in a first-class manner.

When FUSE encounters uncertain control flow during symbolic evaluation, FUSE *generalizes* the effects of different possible paths through the program to create a conservative approximation to any possible path taken at run time. Function calls in FUSE are transparently evaluated, except where the arguments to the current call are a sufficiently close (and safe) approximation to the arguments to a previous call as to allow safe reuse of the results of the previous call (and reuse of the specialization for that call) without any significant loss of run time efficiency (i.e., without loss of ability to additionally specialize that function for the given call). As a result, FUSE essentially adheres to a "greedy" strategy for creating and reusing specialized functions – an advance that represents a large step beyond the trivial overspecialization performed by other other specialization systems.

While FUSE's philosophy permits failure of the specializer to terminate on programs which will themselves terminate at run time, FUSE guarantees specializer termination in other cases by detecting cases *speculative* looping during symbolic evaluation (in which a piece of code is repeatedly evaluated without knowledge that it will be executed at run time). Rather than continually unfolding the recursion or creating specialized functions for each possible recursive call, FUSE breaks the recursion by repeatedly generalizing the results of each loop iteration until a fixed point expressing the results of the loop is reached, and creating an increasingly less specialized function for each such generalization. Once the fixed point is reached, the "function cache" is guaranteed to contain a specialized function of sufficient generality to be used by all subsequent iterations of the loop, and a residual call to that function is emitted. This termination procedure is considerably stronger than any seen in other online partial evaluators, and provides an effective (if somewhat imprecise) means of permitting recursion when it is known to be required, but preventing useless specialization or computation when it is unclear that the run time behavior of the program will benefit from it.

Weise's FUSE system represents the most complete and practical existing online specialization system yet created. PARTICLE adapts a number of ideas from its design for use in the imperative domain.

2.2.1.2.3 Summary of Research in Online Evaluation By discovering opportunities for program specialization during full-bodied symbolic evaluation of the input program, online partial evaluators are capable of discovering classes of specializations that are beyond the reach of offline evaluation, can offer simple termination guarantees, and allow the easy sharing of specialized functions among many call sites and run time contexts. Unfortunately, the need to symbolically evaluate the entire input program and to simulate all aspects of system state is associated with significant compile time performance penalties, and the price of decreased run time for the specialized program is a considerably increased compile time.

2.2.2 Partial Evaluation: Conclusion

The sections above characterized two major frameworks for specialization of programs with respect to static data. Offline techniques offer much faster specialization and offer a possible means of automatically generating compilers and compiler generators from a partial evaluator and a programming language interpreter through repeated self-application. Online evaluators perform a more thorough form of program analysis, and are capable of implementing a proper superset of the specializations possible under offline evaluation. Unfortunately, the conceptual simplicity and thorough character of online evaluation can lead to very long compilation times for the specialized programs.

Having briefly described the partial evaluation literature, it should be noted that almost all of the partial evaluation work has been applied within the context of functional languages; the automatic partial evaluation of imperative and low-level languages has remained an open area of research for the past two decades [28][22]. While some preliminary theoretical work concerning the specialization of existing imperative languages has appeared within the past decade [17][39], it is only in recent years that partial evaluation techniques have truly been brought to bear upon the area of traditional and low-level imperative languages. In particular, substantial work at DIKU has studied the application of offline partial evaluation techniques to imperative and low-level languages, and has included the construction of a partial evaluator for C.[51]. The author is not aware of any existing systems that apply automatic online specialization techniques to such languages.

2.3 Abstract Interpretation and Symbolic Evaluation

Abstract interpretation [1][12] of a program involves the stylized execution of a program, but in a non-standard set of domains. Rather than making use of the standard "concrete" domains (machine values and stores) normally associated with the program execution, abstract interpretation evaluates a program in a set of "abstract domains" that represent *approximations* to the concrete domains. Typically the choice of particular abstract domains for use in a given abstract interpretation problem depends strongly on the type of information that one wishes to collect about program behavior, but the domains and the abstract interpretation procedure will also be formulated in such a manner as to provide a *safe* approximation to characteristics of the concrete domains at run time and to guarantee termination of the program analysis for any input program. In addition, the desire to perform rapid abstract interpretation often leads to abstract domains and abstract interpretation which are very simple and specialized in structure and are aimed at sophisticated collection data just about some small aspect of program behavior (e.g., collecting data about the set of possible referents for pointers or the set of program quantities associated with constant data in a program) [15][2][1].

Although the algorithms may not be explicitly phrased in terms of abstraction and concretization functions, abstract interpretation has been used or proposed in many areas of program analysis: Many traditional data flow and control flow analysis algorithms can be implemented as abstract interpretations [1], and algorithms for strictness analysis [1], in-place update analysis [1], reference counting [27], and pointer analysis [8][26][15] have been formulated explicitly within the framework of abstract interpretation.

In [57] Weise notes a close similarity between the process of symbolic interpretation and program analysis via abstract interpretation, and observes that symbolic evaluation can be characterized as "abstract interpretation with concrete values". From this perspective, symbolic evaluation falls within the abstract interpretation rubric, but is associated with a rather unusual set of abstract domains and abstract interpretation functions. In particular, while most abstract interpretations are performed using abstract domains that are vastly smaller than concrete domains, and represent idealizations of specific aspects of these domains, symbolic evaluation is associated with abstract domains that represent *extensions* to the concrete domains. In some symbolic evaluation systems, the extensions are rather minor: the concrete domains need only be represented by a $\top$ ("top") element representing an unknown value, and possibly by a $\perp$ ("bottom") element explicitly representing a non-terminating computation as well. In more sophisticated symbolic evaluation strategies (such as those used in REDFUN-2, FUSE, and PARTICLE) considerably more additional structure is added in the form of subdomains involving set of possible values, ranges of values, bottom elements of typed subdomains such as "number", pointers, etc.

Just as the abstract domains associated with symbolic evaluation represent a significant expansion and generalization of the concrete domains that obtain at run time, the abstract interpretation associated with symbolic evaluation typically represents a significant generalization of the concrete evaluation procedure (and includes the concrete evaluation as a simple special case).[5] In its use of a generalized evaluation function, symbolic evaluation contrasts very strongly with the greatly simplified abstract interpretations associated with other frameworks (e.g., see [8][26][29]). As an abstract interpretation strategy, symbolic evaluation is particularly unusual in permitting the direct simulation of loops, and in directly simulating the effects of function calls rather than treating function boundaries as "firewalls" for analysis. each of these relaxations permit symbolic evaluation to collect substantial amounts of data on program behavior that is not available during traditional program analysis; unfortunately, under cer-

[5]Note that for systems such as PARTICLE which provide guarantees of specializer termination under *all* conditions, this statement is not quite accurate – it is not possible to simulate the run time behavior of a non-terminating program under such evaluators, although the concrete execution of the program can be simulated to any arbitrary degree of precision.

tain conditions they can also be associated with extremely high costs in terms of compile time performance.

2.4 Compiler Optimizations and Program Specialization

It is worth pausing to consider the place of program specializations within the space of all program optimizations, and the conceptual role played by optimizing compilers. Program specializations are transformations that *simplify* pieces of an input program based on knowledge of the set of possible run time values associated with program quantities. Specializations typically involve the elimination or simplification of program constructs in manners that are known to be legal due to particular characteristics of program evaluation, and can be performed at many different levels. Each of the levels of specialization is briefly described below.

2.4.1 Levels of Program Specialization

2.4.1.1 Expression Level Specialization

At the lowest level, individual expressions in the program can be specialized to the characteristics of the run time environments in which they will be executed. For instance, in cases where a variable is known to always hold the same statically known value, constant propagation may be performed and result in the replacement of a given variable reference by a simple constant. Similarly, copy propagation may result in the replacement of a reference to one program quantity by a reference to another program quantity known to have the same value for all possible executions of this expression Constant *folding* and other forms of algebraic simplification can be used to simplify expressions that are known to evaluate to constants, and effectively shift such computations from run time to compile time.

2.4.1.2 Statement Level Specialization

At a slightly higher level, statements can be specialized through optimizations that involve the elimination of predicate tests and the consequent or alternate branches for known conditionals, the unrolling of loops (particularly when knowledge of the number of iterations to be executed at run time permits elimination of the predicates associated with each iteration), and the process of dead code elimination (the "specialization" here taking advantage of the fact that given the characteristics of program behavior, this statement has no possible effect upon program operation, and can be "specialized" to the equivalent non-existent statement.) Specialization of statements can save space, eliminate run time overhead (such as that associated with predicate or loop iteration tests), and can substantially enlarge basic block size.

2.4.1.3 Function Level Specialization

At the next level up, individual functions can be specialized to the contexts and patterns of data usage of particular call sites or sets of call sites. There is frequently there is a large amount of static information available about the actual parameters to function calls in programs [19]; by taking advantage of this information, the run time cost of the function being called can sometimes be enormously reduced. Depending on the tradeoffs desired between space usage and run time performances, such specialization can be performed at different granularities with respect to the number of call sites associated with each specialization. One simple but highly restrictive function specialization strategy involves the specialization of a given function to the run time contexts of *all call sites in the original program.* Under this strategy, functions are specialized with respect to the patterns of usage of the program as a whole; in order to represent a legal specialization for a function, the specialization must be legal given the patterns of data usage in all call sites in the entire program. More fine-grain strategies specialize separate instances of a given function to take advantage of different patterns of usage seen at different sets of call sites. Such specializations exploit the realization that function interfaces frequently offer a generality that is exploited *between* call sites, while each individual call site or sets of call sites frequently just takes advantage of a small fraction of that generality. This technique allows the creation of specialized versions of the general function tailored to exploit the particular needs of the individual or small sets of call sites. Specializing the function in this manner may allow considerable interpretive run time overhead to be eliminated and may permit the shifting of significant amounts of computation from run time to compile time by performing at compile time computations whose result is statically known. In certain cases, the replacement of a general function by a set of highly specialized functions can also allow significant cost savings to be realized due to the elimination of code unneeded in the particular cases used by the program. (For example, consider a large and general print formatting function specialized into two distinct functions associated with two particular formatting specifications. While each of the functions may duplicate a slight amount of code to actually output the specified information, all of the code associated with the parsing of the formatting specification will be eliminated from each of the specialized functions – yielding two specialized functions whose combined size is likely to be considerably smaller than the size of the original function.) Just as was the case for lower-level optimizations, however, there is a point at which further specialization will yield gains in run time performance, but only at the cost of additional space usage by the program. The choice of an optimal set of specializations to perform depends upon the time/space tradeoffs acceptable to the user.

2.4.1.4 Program Level Specialization

At the highest level of granularity, an entire program can be specialized with respect to a particular pattern of input data. The motivations and tradeoffs involved in specialization at the program level are very similar to those at the function-level: Specialization with respect to particular patterns of usage may allow the specialization of general mechanism to special cases, permitting significant run time overhead to be eliminated or shifted to compile time. Just as there was a potential space cost associated with function specialization on account of the need to create several different versions of a single original function for different patterns of usage associated with different call sites, there is a potential space cost associated with program specialization: The space cost of storing several different executables specialized with respect to different input conditions. In cases where the general patterns of usage are sufficiently homogeneous to allow a few program specializations to be sufficient for all anticipated usages, this space cost may be nonexistent or relatively small.

2.4.2 The Role of Existing Compilation Technology

Existing aggressive optimizing compilers typically perform a variety of powerful optimizations. This section discusses the relation of these optimizations to program specialization.

2.4.2.1 Optimization Technology: Specializing and Non-Specializing Optimizations

2.4.2.1.1 Introduction Although they have substantial overlap with the set of optimizations employed by existing aggressive optimizing compilers, optimizations resulting from specialization (referred to as "reductive" optimizations within this thesis) are neither a superset or subset of optimizations employed in such systems. For example, classical expression optimizations that might be considered reductive in nature would include constant folding and propagation [2] and copy propagation [2]. The collapsing of conditional branches [53] and loop unrolling [2] illustrate statement-level specialization strategies employed in traditional compilers. Other optimizations involving optimizations such as common subexpression elimination [2], software pipelining [34], and forms of code motion [2] cannot easily be classified as specializations.

2.4.2.1.2 Reductive Optimizations Conceptually, reductive optimizations are extremely simple: They consist merely of taking existing code and mechanically transforming it in a way that mirrors the process of evaluation. Intuitively, one can think of reductive optimizations as performing one or more steps in a mechanical symbolic evaluation process (as might be dictated by the operational semantics for a language) and outputting the result. For example, one of the more important optimizations performed by PARTICLE is loop unrolling.[6] In this optimization, we are just mechanically "desugaring" a loop into an *if* statement that internally includes the original loop body (which will become a specialized version of a particular loop iteration), followed by the text of the original loop construct. Another reductive optimization commonly employed by PARTICLE is constant folding: The replacement of some expression by its computed result. In this case, the symbolic evaluation simply executes the construct at compile time and translates the resulting value into an expression.

2.4.2.1.3 Non-Reductive Optimizations The examples listed above are just as small sampling of the broad category of non-reductive optimizations that is not currently exploited by the PARTICLE system; such optimizations typically take advantage of certain special characteristics of the constructs or operators in the program text itself; they transform pieces of code in such a way that the behavior of the resulting code matches the behavior of the original code, but in the process may drastically change the series of "reductions" that might be followed for the evaluation of the code. In this process, non-reductive optimizations may introduce new constructs or operators that are in no way directly implied by the original code, and transform the code in a manner distinct from the transforms normally associated with operational evaluation. As an example of an non-reductive optimization, consider code hoisting from within a

[6]Note that as discussed below, the primary importance of loop unrolling probably comes not so much because of its elimination of branches or conditional checks but because of the extra opportunities for specialization offered by "spreading out" and making distinct several different iterations of the loops.

loop. Here we detect that some piece of code is "loop invariant" (and is thus unchanging over the execution of the loop), and remove it from the loop body. This recipe does not fit in well with the model of reductive optimizations as simply performing steps in the process of reducing a construct.

While PARTICLE presently is equipped with virtually all of the machinery necessary to perform many non-reductive optimizations, for three reasons it currently does not perform any optimizations in this category. First, PARTICLE is designed as a system to explore the benefits of performing high quality program specialization. Most non-reductive optimizations cannot really be legitimately viewed viewed as instances of program "specializations". Secondly, support for non-reductive optimizations would frequently need to be "shoehorned" into PARTICLE's system of symbolic evaluation, while bookkeeping for reductive optimizations is easily and transparently handled in the process of expression and statement evaluation (see Section 4.2.4.1 for further details on this process.) Finally, because PARTICLE is implemented as a source-to-source transformation system, it need not take explicit advantage of non-reductive optimizations to generate high quality executables. Such optimizations can be delegated to postprocessing by an aggressive compiler.

2.4.2.1.4 Lack of Higher-Level Specialization At higher levels of specialization, existing compilers are extremely weak in exploiting static knowledge of program data. The author is not aware of any existing compilers that provide any opportunities for specializing a function to make use of patterns of usage associated with different sets of call sites, or any compilers which permit the specialization of a program with respect to external input data for that program.

As was discussed in section 2.4.1 and as will be further analyzed in section 3.1.3, there are tremendous opportunities for improving program performance via specialization at the function and program level. Because external input frequently has global implications for program behavior, specialization at the program level can frequently allow significant optimizations throughout the body of a program and can thereby lead to dramatic performance improvement.

Existing compilers adhere to a policy in which every function in the program source is associated with one and only one function in the program executable, and are thus incapable taking advantage of opportunities for function-level specialization. The primary difficulties with performing optimization in the absence of function-level specialization are reviewed below.

As will be noted in section 2.4.2.2.2, even highly aggressive compilers general fail to propagate rich information on an interprocedural level. Unfortunately, even in cases where interprocedural analysis is performed to allow optimizations in callee procedures (such as parameter alias information propagated in [44][36][50]), such information is computed under the highly restrictive assumption that a particular function in the source code is compelled to correspond to a single function in the program executable (and thus that no functional specialization is possible). As a result, the program is forced to create a single function capable of being safely used by *all* call sites, and will be unable to take advantage of specializations applicable only to proper subsets of call sites. (As will be emphasized in the following chapter, the fundamental issue governing the safe use of an optimization is that it is legal for *all possible executions* of the code.) Even in the presence of very rich interprocedural data flow information, without function-level specialization the performance advantages offered to the callee can be enormously diluted. Even if the value of many parameters or other aspects of the calling context are statically known for *each of the call sites* individually, it is frequently the case that little advantage can be taken of such information, because the specializations applicable to certain subsets of the call sites (e.g., constant folding and loop unrolling made possible by knowledge of parameter values, elimination of unnecessary conditional tests and/or interpretation overhead, etc.) may not be applicable to *all* call sites, and thus cannot be safely performed on the single residual function.[7] This condition can severely restrict the range of optimizations made possible by interprocedural flow analysis. In order to relax this condition, we need to propagate interprocedural flow information (in order to decide in what manner it will be advantageous to specialize a function based on the behavior of the call sites), but the compile time expense of such flow analysis may well pay for itself in the high quality of the resulting specialized code. In short, rich interprocedural flow analysis may be highly profitable for uniprocessors if we permit *function specialization* to take advantage of the information collected by flow analysis.

In fairness to existing compilers, it should be noted that there is *some* capacity for specialization at the function-level that is exploited by existing compilers in the form of inlining (or "open coding"). By incorporating the body of the callee function into the code of the caller, one can both eliminate the call overhead and safely specialize the called function body to the static information known to obtain at the call site. Unfortunately, such inlining is not fully applicable to recursive function, and can require enormous space overhead by the creation of a specialized copy of the called function for each call site at which we inline [44]. In addition, while the incorporation of these specialized functions into the code of the calling routine eliminates the overhead associated with the actual transfer of control and context, and specialization may allow the a significant decrease in the actual size of the inlined code, inlining prevents the *sharing* of the specialized code between many different call sites. As a result, the cost of representing a new specialization cannot be amortized over the performance savings gained from a number of call sites. This requirement prevents the cost-effective creation of specialized functions which exploit similar (or roughly similar) patterns of usage obtaining at several distinct call sites, and imposes an artificial space penalty on the use of specialization.

Function specialization represents a more general and cost-conscious attempt to take advantage of shared commonalities in calls to a given function. While in inlining the specialized function be used by at most a single call site, function specialization allows arbitrarily many call sites to make use of the same specialization. By carefully weighing the overall time savings to be gained by implementing certain optimizations with the (variable) costs to be borne by creating a new specialization, a program specializer can intelligently partition calls originally made to a particular function into sets using distinct specializations of that function. If judged desirable, inlining of the specialized functions can be performed as well. In this manner, general function specialization avoids the high cost overhead of inlining while retaining its fundamental benefits – the specialization of the callee in such a manner as to take advantage to the most important specializations allowed by the contexts associated with call sites.

[7]Note that this problem is slightly exacerbated by the lack of rich domains in the program analysis – were we to have a representation for static values that allowed a more precise specification of our knowledge about static values than the traditional flow-analysis dichotomy between *completely known to be* or *completely unknown*, we might be able to more precisely capture the set of static values that we could possibly receive from different call sites. While such a representation might not allow us to perform certain optimizations, others might be enabled. For instance, knowledge that a certain program value is a member of some set (presumably of cardinality greater than one) would not permit the use of constant propagation on that value, but might permit a conditional to be collapsed, a conditional test eliminated (if it simply tested a condition which all of the values in the set satisfied), or even allow a degree of loop unrolling (by an amount of the smallest member of the set).

2.4.2.2 Analysis Technology

2.4.2.2.1 Analysis Costs of Restricted Abstract Domains and Incomplete Memory Models The program analysis performed in existing compilers is typically fast but is performed using relatively impoverished abstract domains [53], so that such systems may miss even relatively simple opportunities for optimization, as are documented in section 3.3.2. In fairness to some existing systems, highly aggressive compilers frequently employ much richer abstract domains to model the values associated with quantities such as pointers to which program analysis is particularly sensitive [2]. Although compilers that adopt this approach are not capable of discovering some of the optimizations that can be exploited in the presence of richer domains, this approach voids the need to make disastrously conservative assumptions regarding the possible effects of actions such as pointer writes, and thus addresses one of the major shortcomings of performing program analysis in the presence of small domains.

A second source of imprecision in the program analysis performed by existing compilers stems from the failure to model program state in a truly fine-grained manner. In particular, even aggressive compiler systems often make no attempt to keep track of the values associated with individual elements of an array, and are incapable of performing constant propagation on the values of array elements; for the sake of dependence analysis such systems typically conservatively treat reads and writes to particular elements of the array as references to the entire array.[8] While the machinery and overhead involved in modeling the run time state at such a fine granularity can be substantial, it allows array elements to be treated as first-class elements of the program state, and when coupled with a capacity for high quality specialization and the ability to propagate values on an interprocedural basis can permit significant exploitation of statically known data in scientific code (e.g., by permitting the specialization of an array multiply routine with respect to a single known argument, or the specialization of a filter program with respect to a particular filter). The additional presence of rich domains for general program values (including array indices) can permit correspondingly more precise dependency analysis, and a more precise bounding of the effects of array writes.

Although performing abstract interpretation using rich abstract domains can allow the discovery of optimizations inaccessible to systems maintaining less precise knowledge of program values, it remains to be demonstrated whether the resulting performance advantages are significant for most programs.

2.4.2.2.2 Analysis Costs of Restricted Abstract Interpretation Even when a compiler's static analysis mechanism has the capacity to exactly deduce the run time value of program quantities, the abstract interpretations carried out by existing compilers is sometimes too weak to permit effective leveraging of this information in further analysis or to make thorough use of this information in even statement or expression level specializations. Large shortcomings in the quality of traditional compiler analysis and optimization are particularly possible in the two major areas in which the character of full-bodied symbolic interpretation differs the most

[8]I have been told that a highly optimizing compiler for FORTRAN H did not conform to this tradition, and was capable of entirely evaluating an FFT routine involving known arguments at compile time; such a compiler seems likely to have made use of a technique very similar to symbolic evaluation in that it must have been capable of simulating the effects of program loops on an iteration-by-iteration basis wherever allowed by static information [14]..

from traditional analysis methods: the treatment of loops and function calls.

2.4.2.2.2.1 Loops Traditional compilers approximate the effects of loops by one-time combination of synthesized information characterizing the behavior of the loop with inherited information [2], or (more generally) by resorting to the construction of a fixed point bounding loop effects over all possible iterations of the loop [2]. Such methods typically yield very fast loop analysis with a minimum of overhead, but fail to take advantage of additional information that would allow more precise loop analysis, such as static information that could be used to discover the number of iterations associated with a given execution of a loop.[9] In contrast, symbolic evaluation is capable of performing extremely fine-grained loop analysis whenever possible and deemed desirable. In particular, wherever static knowledge about program control flow is available, symbolic evaluation adheres to this information, allowing loop analysis to proceed on a per-loop iteration basis. While such analysis can be far more expensive than simple fixed point iteration, it avoids a tremendous loss of information than can arise in the latter methods as a result of the need to specialize the loop on the basis of an entry state that is simultaneously a legal approximation to the entry state after an arbitrary number of iterations. As discussed in section 4.3.3.2.2.3, the construction of such an approximation "mixes together" the beginning state of the loop in a manner that prevents can prevent recognition of important invariants of the data and inhibit associated specializations. Figure 2-3 provides a simple example of a case where where existing compiler technology is incapable of effective exploitation of static data in loops despite enormous potential for optimization, while the more exact analysis possible during symbolic evaluation permits the use of such data in program optimizations.

From a global perspective, the opportunity for performing fine-grained loop analysis can lead to a program analysis strategy that better reflects the fact that typically the tremendous majority of the run time of a program is spent within a small fraction of the code. By offering the option of analysis program loops in a far more precise manner than is possible using traditional compiler techniques symbolic evaluation permits a concentration of compile time resources in areas where they may be the most needed.

2.4.2.2.2.2 Function Calls The analysis of the effects of function calls in traditional compilers is prone to the same general weaknesses as loop analysis in such systems. Because function boundaries represent "firewalls" to program analysis in traditional analysis, existing compilers must make conservative approximations to the effects of particular function calls, and are incapable (except through the mechanism of inlining) of precisely discovering the return value or state of a function call even when it is statically determinable. Aggressive interprocedural analysis techniques such as those described in [45] and [44] collect information on program use and modify information at the interprocedural level; this information permits the adoption of far less restrictive assumptions to be made about possible effects of procedure calls, but does not offer the possibility of optimizing the callee based on knowledge of parameter values, of

[9]Note that while some compilers are capable of extracting information concerning the number of run time loop iterations that will be executed in certain very restricted circumstances (such as instances where the loop bounds are explicitly given), such ad-hoc methods are not of general applicability, and are typically very sensitive to slight perturbation in the structure of the program construct being analyzed.

```
int strcmp(char *sz1, char *sz2)
{
  for(; *sz1 == *sz2 ;  sz1++, sz2++)
    {
    if (*sz1 == '\0')
        {
        return(0);
        }
    }

  if (*sz1 > *sz2)
    return(1);
  else
    return(-1);
 }


...


/*  str1, str2  statically known */


if (!strcmp(str1, str2))
        {
        ...

        }
else
        {
        ...
        }
```

Figure 2-3: The code for *strcmp* is completely statically computable (and reducible) in cases where the argument strings are statically known. Traditional compilers do not take advantage of such information.

exactly deriving the return value of even the simplest a function call or of precisely modeling the execution state after such a call even where such information are statically determinable. Such limitations impose performance penalties on the use of functional abstractions by hampering analysis and low-level specializations in the presence of function calls, and prevents the detection and "constant folding" of pure function calls yielding statically known values.

By contrast, symbolic evaluation transparently crosses function call boundaries, and tries to create as precise a model of the return state and value of a function as is possible with the available static information. Some online systems [56] provide a limited mechanism for memoizing return values of function calls (in a functional language) to avoid the overhead of repeated symbolic evaluation of a call with the same static information available, but in online evaluators the overhead associated with approximating function calls tends to be far greater than is the case in traditional analysis. This fine-grained analysis of call behavior can allow exact deduction of return value and return state for a given call, permits precise recording of the specializations in the called function that are legal for a particular call, and places the analysis of code in called functions on a truly first-class setting with the analysis of code within a given function.

Recent studies have made important contributions in suggesting that *traditional* interprocedural analysis offers little benefit for improving code code on uniprocessor code [58][45]. However, even the most aggressive interprocedural analysis schemes in traditional compilers for imperative languages are used only to collect information on the possible program quantities associated to and used by the called procedure (as well as any information needed for determining parameter aliases), and typically focus on the benefit of such analysis to the optimization of the caller on account of the need for making less conservative and restrictive assumptions about the effects of the function call upon system state [45][44]. Even highly aggressive interprocedural analysis typically collects rather little data relevant to the optimization of the *called* procedure (interprocedural alias information is one important exception) [44]. At the same time, heuristic evidence suggests that substantial optimizations in the called may be possible by making use of interprocedural flow analysis information; a study of PL/I programs showed the 24% of all function parameters are simple lexical constants [19]; the potential for the use of such static information is great.

The character of interprocedural analysis carried out by symbolic evaluation is very different from the traditional forms of such analysis, and is aimed particularly at take advantage of the potential for optimizations of both the callee and caller enabled by static knowledge of parameter values. The information propagated by interprocedural analysis during symbolic evaluation is used to carry out specializations in both the the caller and callee function, and is far richer than that propagated during traditional interprocedural analysis: It involves complete approximations to system state, rather than conservative approximations to reference and use information. While the reference and use information collected during traditional interprocedural analysis can be used to *rule out* illegal "optimizations" of the caller and (when using alias information) of the callee that were discovered by local analysis, the information collected during the full-bodied interprocedural analysis performed during symbolic evaluation can be used not only to rule out the same set of illegal "optimizations", but also to *discover* a new set of legal specializations both in the caller and callee.

2.5 An Overview of PARTICLE

The bulk of this chapter has been spent discussing previous work related to program specialization and symbolic evaluation. This section provides a brief high-level characterization of the PARTICLE system and places PARTICLE in the context of this other work. Chapter 3 provides a discussion of the motivations and tradeoffs considered in the design of PARTICLE, while chapter 4 turns to examine the abstractions and mechanisms employed in PARTICLE. Thus, chapter 3 examines the "why" of PARTICLE, while chapter 4 considers the "how."

PARTICLE is a fully automatic polyvariant partial evaluator for a powerful imperative and low-level language (a slightly restricted subset of C). PARTICLE operates by means of symbolic evaluation, and manipulates values drawn from an extremely rich and arbitrarily extensible abstract domain. Like existing online evaluators for functional languages, the system shares makes use of full-bodied symbolic evaluation to discover possible program specializations, but delays committing to specializations until after symbolic evaluation has terminated and all data concerning program behavior has been collected. This decoupling of evaluation and specialization decisions permits extremely high quality specialization decisions to be made using explicit space and time cost/benefit information collected over the evaluation of the entire program. PARTICLE operates by means of source to source transformation, and is designed to complement (rather than replace) traditional aggressive compilers. Although PARTICLE is a fully automatic system and requires no user direction or intervention, PARTICLE provides the user with the option of exercising a wide degree of control over the scope and character of the specialization process, permitting a series of tradeoffs between specialization quality and compilation time. PARTICLE has been applied to a variety of small programs, sometimes yielding remarkable speedups over code optimized by traditional aggressive compilers. However, the full-bodied character of the symbolic evaluation carried out by PARTICLE makes it extremely slow, and of questionable applicability to large systems.

Chapter 3

PARTICLE: Tradeoffs and Design Decisions

This chapter expands upon the motivations for program specialization, discusses the use of arbitrarily extensible and high quality abstract domains in program analysis, the advantages arising from arbitrarily precise simulation of control flow, and the enforcement of termination guarantees.

3.1 Motivations for Judicious Program Specialization

The general motivations for program specialization were briefly sketched in the introduction to this thesis. This section discusses these motivations in greater detail. Some discussion is also provided for cases in which program specialization would bring *no* benefit.

3.1.1 The Need for Specialization

3.1.2 General Advantages of Specialization

Engineers often prefer to design software at a relatively high level of abstraction, preferring the use of general black box abstractions to the tedium and danger of hard-coding code to take advantage of special cases into the program. Taking explicit advantages of "special cases" of general interfaces frequently requires substantial knowledge of their internal structures, in order to determine which cases are worth pursuing. Obtaining this knowledge requires violating the "black box" presented by an abstraction and relying on internal properties of their implementation. This violation of abstraction can make code maintainance and modification tricky, and can easily lead to programs that run fast but incorrectly. By automatically recognizing and exploiting special cases in a safe manner, program specialization can void much of the need to manually tailor the program to exploit such cases and can thereby substantially lessen the performance penalties associated with adherence to abstraction in program design.

Even in those projects designed by performance-hungry engineers who have the necessary skill (or foolishness!) to abandon the use of abstraction barriers and create programs tightly coupled to the particular structure and type of data being manipulated there can arise a need for program specialization. At some point, a program simply becomes too large for even the most experienced set of engineers to to play the game of "logical brinkmanship" to the extreme, and the potential and desire for specialization begins to appear.

To provide a concrete example of an instance where abstraction can be needlessly associated with a performance penalty, consider an inner loop of a function where the loop body invokes a routine to draw polygons at different three-dimensional depths in the central portion of a window. Now consider this function applied to data that describes non-overlapping polygons. For each iteration of the loop, the routine to render the polygons will check to see if portions of each polygon need to be clipped, and whether it overlaps portions of previously drawn polygons. While this is clearly an undesirable and possibly highly substantial performance overhead, the software engineer cannot easily eliminate it without going through the trouble of creating a specialized version of the rendering routine that handles the very specialized case of non-overlapping and non-clipped polygons. While the need for higher performance may drive some engineers to perform this manual specialization of their code, such a change will make the program less maintainable and more error-prone. By duplicating the functionality of the original function (even for a restricted range of inputs), the engineer creates additional routines that must be changed in the case of program modification. Moreover, while a call to a specialized routine may be initially made within a safe context, the patterns of data at that point in the program may later change in such a way as to no longer be amenable to using the special case. At the least, the software engineer must remember to return to the point and instead invoke the more general routines; in unfortunate cases, the dependence of the code upon the particular pattern of data will be forgotten and a bug will result.

As programs grow in size and complexity, the need for abstraction software development becomes ever greater – both because widespread violation of abstractions becomes increasingly dangerous, but also because engineers are less likely to know the system being designed thoroughly enough to be able to recognize cases where special cases can be exploited [38][24]. At the same time, the performance costs associated with these abstractions become ever more weighty. Program specialization systems offer the potential for automatically and substantially reducing the performance cost of this abstraction, permitting the construction of highly abstract software while avoiding the performance overhead traditionally associated with the use of abstraction.

3.1.3 Exceptional Specialization Opportunities

While the widespread use of abstract interfaces make program specialization desirably in a wide variety of circumstances, there are a number of commonly contexts in which the use of program specialization can be unusually advantageous. A few of these are common enough to deserve special mention.

3.1.3.1 Cases where Static Data is Generally Known

In certain cases, it is common for general purpose functions (or other mechanisms) to be applied to data that is at least partially known. In these cases, the full generality of the abstractions is typically exploited *between* different call sites, rather than within different instances of execution at a particular call site (i.e., the flexibility is exploited in the spatial rather than the temporal dimension). Several examples will serve to illustrate the general phenomenon.

Consider the function *printf* from the standard C I/O library. In the vast majority of uses of this function, the formatting string passed to the routine is known at compile time. While it is perfectly possible to pass different strings to the function from different executions of the function from the same call point, the routine is almost always used in such a manner that the formatting string (although not other arguments) are exactly known at compile time.[1]

Similar patterns of use occur in abstractly written scientific code. (Compendiums of numerical and scientific algorithms such as [43] offer a variety of examples of this phenomenon). Consider for example, the routine *bessj* from [43], which evaluates the Bessel function $J_n(x)$. This routine accepts two arguments: The index n of the bessel function to be used, and the point x at which to evaluate this function. In many instances (perhaps the substantial majority), we can expect that the particular Bessel function to be used *at a particularly call site* will often be static and known in advance, as it may depend simply on the particular normal mode one is interested in within the physical situation being modeled. As seen in figure 6-4 static knowledge of the particular Bessel function to be used allows substantial simplification of the code for that function, and allows considerable speedups.

As a third example of this phenomenon, consider a routine used in a clustering analysis program that calculates the volume of a hypersphere of a variable dimension and radius. In clustering analysis problems, the user will frequently the dimensionality of the data to be clustered (and thus the dimension of the hypersphere) long before the data itself is available, and a specialized version of the program can be constructed in which this dimensionality is specified (in this case, we are using the program specializer particularly as a partial evaluator). All calls to this function within the program will be associated with the same fixed dimension but with a variable radius. In the volume formula, many constants are purely static for a given dimension, including calls to expensive math functions such as those evaluating the gamma function and raising one floating point value to a floating point power, and such calls can be entirely evaluated at partial evaluation time and reduced to simple constants. In this way, significant amounts of run time overhead can be eliminated.

Finally, at a somewhat higher level of abstraction, consider a class abstraction that implements a single level in a hierarchical pyramid of images of increasingly finer approximations along with mechanisms for correlation-based matching. (Such pyramids are frequently found in image registration contexts, where matching between smaller and smaller pieces is performed, with each level making use of "hints" from the larger context in which it is contained). While the image data itself may not be available until run time, frequently much of the data determining the behavior of class methods at given level is known at compile time. Such information can allow substantial specialization of program structures. For example, the bounds and dimensions of the pixel maps and the size of the "bounding box" to search are statically known at each level, permitting complete compile time evaluation of the FFTs of known mask data (such as masks of "all ones"), partial computation of the FFTs of other data (due to zero-padding of known size), and allowing the unrolling of many loops. Thus, while the actual data to needed perform the matching process is not available until run time, there are a variety of static parameters known at compile time whose careful investment can yield strong performance enhancements.

Each of these cases illustrates an abstraction which offers substantial benefits for software engineers by permitting the use of shared, general, mechanisms for a variety of conceptually

[1]As will be seen below, specializing common uses of *printf* with respect to known formatting strings can offer particularly substantial performance payoffs because of opportunity for interpreting the string at compile time, and thus shifting the associated overhead (string scans) from run time to compile time.

similar situations. However, such mechanisms are typically used in a variety of cases with different *statically known* parameters. These cases represent classical examples of where program specialization can offer substantial performance benefit.

3.1.3.2 Dynamic Method Dispatch

Object oriented languages such as SELF, SMALLTALK, and C++ offer type-based method dispatch at run time. In these languages, the type of the object being manipulated can determine which particular code is invoked for a specific method (class function) invocation. Code for such languages commonly offers in which the run time types of objects can be discovered statically (either because they have been explicitly checked by the user, checked implicitly by previous operations, or because they are manifest from context), but where such information is beyond the reach of traditional compilers. In software systems constructed within such languages (or within oo-based environments written in other languages), program specialization methods offer the potential for yielding substantial additional performance improvements: Function invocations that naively require implementation as indirect calls (calls after function lookup) but whose targets can be statically determined can be specialized to call directly to the appropriate routine or (frequently even more desirable) directly open coded by the compiler. Since type dispatch occurs with high frequency in object oriented systems, the ability to eliminate the abstraction, indirection, and branching overhead of the function dispatch can be very important.

3.1.3.3 Interpretive Mechanisms

In the presence of interpretive mechanisms, frequently even partial knowledge of data can permit lucrative specialization. In such cases, run time operation can frequently be conceptually separated into two distinct classes of activity: "pure interpretive" overhead, in which the program classifies the data and decides what action to take on the basis of this classification, and a "performance" step in which the program actually applies this action to the data. In some cases, the "interpretive" step may only be implicit (e.g., consider a program that computes a generalized polynomial from a general array of input coefficients – see figure 6-3.) but the overhead typically remains.

Examples of the interpretive rubric abound, and are found wherever programs deal with abstract classification or data of many different categories: Parsers (where the overall structure is devoted to discovering "what" the input is, and where the productions represent the "usage" step), programming languages interpreters (where one discovers the structure of the each expression or statement and then "does the right thing" for that construct), output formatting routines for programming languages (see below), programs involving the evaluation or interpolation of some user-specified function [21], utilities such as *grep*, *find*, etc. In many such cases, a program specializer can frequently substantially improve performance (and possibly effect space space savings as well) by executing much or all of the interpretation entirely at compile time, and eliminating the associated run time overhead by creating a residual function for particular input that only contains the code needed to process the *unknown data.* Such a strategy is possible even if the action to be ultimately performed after interpretation must be delayed until run time.

For an example of a simple but common interpretive mechanism, consider the general purpose formatting routines frequently offered by languages or language libraries (e.g., *printf* in

C, the *FORMAT* statement in FORTRAN). Such routines are often capable of outputting a range of arbitrarily formatted strings based on some specified format. The formatting specification applied to such routines at a given call site is often (indeed, almost always) static, while the data to output may vary considerably among different uses or call sites. As a result, the formatting specification that constitute the arguments to the routine at any single call site are typically known at *compile time* and it is possible to avoid the *run time* overhead of interpreting that formatting specification by performing the interpretation at compile time. For example, a call to a routine of this sort that involved a format specification telling the routine to print out three integer variables in a certain pattern might be reduced to three run time calls to an integer-printing routine for each of the three variables. For reasons discussed in section 2.4.2, while aggressive optimizing compilers are capable of thorough low-level specialization, such systems would find it difficult to make this reduction, since the interpretation of the format string may be an involved process, requiring a scan or several scans across the string, coupled with some potentially complicated control machinery. The memory model maintained by traditional compilers is not capable of representing strings or arrays, and such systems do not have the capability for performing fine-grained simulation of program loops. As a result, removal of the interpretive mechanism is beyond the bounds of standard compiler optimizations. Within a framework of a program specializer based on symbolic evaluation, however, such a reduction would be made naturally, and the interpretive overhead could be eliminated even if the actual values of the objects to be printed are not known by the partial evaluator.

3.1.3.4 Domain Checks

A particularly common form of overhead that can frequently be eliminated by program specialization is the *domain check* – instances where an abstraction verifies that a given set of data is in some way "acceptable." Strictly speaking, this case is just a trivial example of an interpretive mechanism (where interpretation is only used to distinguish between two categories of inputs), but the *quality of reasoning* needed to deal with these cases is often far simpler than that needed to process more general forms of interpretation, and the overhead associated with general interpretation is often greater than that associated with domain checks (e.g., consider the interpretation of a formatting string vs. a check as to whether an argument is non-negative in a square root routine). As is the case for dynamic type-based dispatch and other interpretive cases, in many cases we can statically demonstrate that the classification of the input data falls within the desired range – whether on the basis of previous implicit or explicit checks, of simple mathematical identities, or because we have been provided explicitly or implicitly with bounds on the range of the data being manipulated. Frequently such demonstration is immediately obvious, given the context in which the domain check takes place (e.g., when one obtains a creates a new array of specified size, and is initializing each argument to some fixed value, no bounds checks are needed); in other cases, it is rather subtle and is considerably beyond the range of any existing specialization system (e.g., a proof that if one searches a binary search tree for an arbitrary key and is unable to find it, then no duplicate checks are needed when subsequently inserting that key in the key).

Because of the simple character of domain checks, the elimination of any particular check is unlikely to have a noticeable effect on program performance. However, when domain checks are invoked with each operation (such as array bounds checks upon array accesses) within tight inner loops, the overall performance savings can be non-trivial (particularly if the "default" behavior consists merely of an operation such as table lookup or the application of a hash function). Moreover elimination of the checks can avoid the penalties associated with taking a branch and enlarge the size of basic blocks. Two examples of domain checks are presented below, one of which makes use of relative simple information available at compile time, and the other of which demands more sophisticated analysis.

For an example of the use of easily obtained information to eliminate domain checks, consider a matrix multiply routine that verifies that the row and column dimensions of left and right matrices are the same. It may well be that because of the character of the analysis one is performing (e.g., 2D image transformations) the size of the matrices being manipulated is statically known. In such cases, the run time dimension checks are completely unneeded and can be eliminated – provided that the statically known dimensions are indeed equal! On the other hand, it may be that the matrix sizes will not be known statically, but that we will *still* be capable of eliminating most of the checks, simply by recognizing that after we have checked them a single time, we need never fear that the dimensions are different.[2]

As a second example, consider the application of bounds checks to array references. As alluded to above, frequently contextual information allows such checks to be be safely eliminated. Array sizes are frequently statically available, and in many cases loop boundaries implicitly guarantee that array indices are within the legal range, and in some cases static knowledge of the array subscript expression (whether precise or less exact) allows verification that the array reference is permissible (e.g., accesses to fixed offsets, or accesses within explicit checks on the values of the index.) In other cases, it is possible to recognize (as above) that once we have checked the legality of a given index a single time, there is no need to do so again – permitting multiple accesses to the same array element with only a single check necessary.

As a third example, consider an inner loop in a program in which the code is calculating the distance between two points in two dimensions. This calculation involves a call to a square root routine with an argument that represents the sum of the squares of the disparity between the coordinates of the two points in the x and y directions. The square root routine checks its argument to verify that it is a non-negative number prior to iterating to a solution. Making the non-trivial assumption that the sum of the squares of the two displacements is guaranteed to be a non-negative number (a fact that may be known to the programmer or a static analysis system due to the ranges of values involved), the domain check (executed once per iteration of the loop) is entirely unneeded and could be safely eliminated. Depending on how the square root is performed, this could lead to a not inconsiderable performance enhancement. While this example has many similarities with those discussed immediately above, note that the reasoning involved is somewhat more sophisticated. In particular, it requires the ability to draw on the mathematical fact that the square of any element in the domain of the x and y

[2]The elimination of the domain checks in this case requires the employment of a more sophisticated means for the collection of static information than is needed for the case where static information concerning matrix sizes is available. In particular, the second case requires the use of "conditional contexts" where information concerning program quantities can be gained by entrance into conditionals that test the values of those quantities. Conditional contexts allow the recording of the fact that the matrix row and column sizes are indeed equal for all code following the first test, information that permits the elimination of further checks checking such information. This mechanism is currently not supported in PARTICLE, although its addition to the system is being considered (see section 5.5.3.

coordinate representations is a non-negative – a fact that is non-trivial both from the point of view of mathematics and of discrete representations to ideal numbers. In the absence of general purpose extensions to allow reasoning in these areas, it is difficult to imagine a truly widespread use of such reasoning.

3.1.3.5 Staged Programs

A final category of cases in which we can expect unusually large performance enhancements are programs in which some pieces of the external input to the program can specified (or partially specified) earlier than others. In such cases, we can create a version of the entire program specialized to the particular set of input that has been partially specified. Conceptually, the benefits accrued by possession of knowledge of particular input data is not qualitatively different than benefits from other static data deducible from or computed by the program text – in both cases, we can use the information to safely specialize the program's behavior, and may originate in through the use of very similar mechanisms (calls to external routines from within the program all look the same to PARTICLE, whether the routines perform i/o or any other function). However, a little thought suggests that while the *qualitative* benefits of such different types of knowledge are no different, the *degree of leverage* we get from an incremental increase in knowledge of input information can often be far more substantial than from a slight increase in knowledge of about some other quantity in the program text. In short, the *performance benefits* of a little knowledge of program input data can in general allow much greater performance optimization than a little knowledge of an internal program value. The reason for this imbalance is simple: Given *all* of the input data, a program specializer operating via symbolic evaluation could perform an arbitrarily complete "specialization" of the program – it has *all run time information* within in its grasp. Unless constrained by user limits on running time and/or user desire to explicitly *avoid* evaluating certain classes of constructs (e.g., I/O constructs), the symbolic evaluation performed by a program specializer given complete knowledge of the program input would actually result in the complete evaluation of the program. Moreover, the whole *purpose* of most programs is to use external data (whether in the form of menu selections, mouse movement, the source of a program, a circuit description, or any other manner of input) to dictate their internal computations. In this sense, input data frequently has global implications for program behavior. On the other hand, no matter *how* much we know about internal program values alone, for almost all programs we will still lack a great deal of information – all the information which is not computable without knowledge of the parameters. Conversely, each small amount of information about an input parameter to the program can have truly extensive and global ramifications about our knowledge of the *internal* values of the program that directly or indirectly depend on the value of that parameter. (Note that precisely the same reasoning can be applied at other levels of abstraction within program analysis, particularly at the function-level – every bit of information we can gain about an input parameter to a function is likely to have many substantial consequences for the behavior of the function. Program specialization makes extensive use of knowledge of input parameters at this level as well.) As a result, by permitting significant portions of the program's computation to proceed at compile time, knowledge of program input can often lead to dramatic speedups.

Staged programs are very common, and it will be worth mentioning a few examples. It is hoped that in each case, the opportunities for performance enhancement should be manifest. Consider the specialization of a correlation-base image pattern matching program with respect to a particular image to match (thus allowing at least one FFT to be eliminated at run time), the specialization of a naive string pattern matching program with respect to a static string (thus automatically yielding the Knuth-Morris-Pratt string matching algorithm for the specified string), the specialization of a circuit simulator with respect to a particular circuit (but leaving the time period at which to simulate unspecified), the specialization of a program that automatically discovers and applies machine dependent optimizations for an abstractly specified computer architecture to a particular abstract specification [42], the specialization of a shape-drawing program to particular shape descriptions (but leaving the particular locations at which to draw these shapes unspecified), the specialization of a language interpreter with respect to a particular program (but leaving the input specified), or the specialization of a neural network simulator with respect to a particular network configuration (but leaving the input vector unspecified). In each of these cases, the opportunity for performance improvement resulting from specialization is great.

3.1.4 Containing the Space Explosion

It was briefly argued in section 2.4.2.1.4 that use of inlining as the sole means of program specialization is in general unacceptable, as it creates a separate copy of the specialized function for each call site being inlined – a practice that prevents sharing and thus prevents amortization of the space increase resulting from specialization over all call sites that make use of the function. Many previous program specialization systems tended to make use of trivial specialization (where new specializations are created for each call site) [10][41], or employ greedy strategies for creating and reusing specialized versions of the original functions [56]. While trivial specialization offers the run time performance advantages inherent in function specialization, the space cost of such a strategy approximates (and may even exceed) that of inlining. In most strategies, this cost is unacceptable. Greedy strategies (where specializations are created on-the-fly and reused where judged possible and not overly inefficient) offer a far more satisfying approach, but are insensitive to global patterns of function use, and may be difficult to effectively throttle (e.g., it is not entirely clear how to modify such strategies to provide the user with explicit control over the space/time cost/benefit ratio). As hinted above, one basis for a more powerful strategy is to recognize the function specialization problem as a *partitioning* problem. Conceptually, a given function is associated with a set of run time calls to that function, each from some particular call site. For each of these calls, there is some particular set of specializations that can are known to be legal. In the creation of a set of specialized functions for a given function, our job is to partition the calls to that function into a set of equivalence classes, where each equivalence class corresponds to all calls to a particular specialized function. This partitioning is performed in such a way as to balance the space cost of creating the new function (which may be of different size than the original function) with the performance advantages associated with taking advantage of as many specializations as possible, subject to the constraint that we cannot take advantage of a particular specialization in a function unless it is permissible in *all* of the calls to that function. For example, if we have a large number of different calls whose contexts (parameters and other aspects of system state) allow a certain set of optimizations in the callee, the partitioning function will try to partition these calls so that they use the same specialized function. Although it is not computationally feasible to exhaustive examine each potential partitioning and find the optimal one, heuristic techniques for combinatorial optimization problems can be used to efficiently provide a good approximation the ideal partitioning. This strategy allows the efficient incorporation of information about global aspects of program behavior (such as estimates of the total number of calls to each function and their associated set of legal specializations for the execution of the entire program, space cost and time/space benefit information associated with each information), and permits explicit guidance by user-specified preferences for time/space tradeoffs.[3]

3.2 Origin and Representation of Static Knowledge

3.2.1 Tutorial: The Character of Static Knowledge

This thesis makes repeated use of phrases such as "the static value of a program quantity", "the approximation to the run time state of a program at point p". It is worth pausing for a moment and reviewing some of the concepts and distinctions underlying such statements and their relation to program specialization. This discussion will reemphasize the important coupling between program analysis and program specialization that can be effectively exploited by program specializers. The material presented within this section is deliberately tutorial in character, and is designed to help reinforce an understanding of the origin and constraints on static knowledge. Readers who feel comfortable with the material are urged to skim or simply skip this discussion.

Consider first the execution of a program at run time. At a given point in the run time execution of the program with a given set of program input, every existing program quantity has some precise value (for the sake of this discussion, a "value" can be thought of as just being used as a synonym for some *particular pattern of bits*). That is, if we were to stop the execution of the program at a particular moment in time during run time execution, every existing program quantity would be associated with some precise value.[4] Thus, at a given point in time during the run time execution of a program evaluation the values associated with program quantities have some precise value.

[3]For the sake of precision, it should be noted that the discussion above glossed over a detail the specialization decisions. We characterized specialization as the partitioning of *calls* to the function being specialized, where the equivalence class associated with each call would dictate the specialized function targeted by that call. (Note that in this context the term "call" refers to the symbolic evaluation of a call at compile time – a compile time evaluation that may itself simultaneously simulate the effects of many actual run time calls.) While such a system is possible in principle, it is far easier (although somewhat less efficient) to partition on the level of *call sites.* If we were to partition at the level of individual calls, we can take advantage of differing patterns of specializations permitted from distinct calls from the same call site (e.g., calls made with different parameters from the same call site within a loop), but this requires some method of "dynamically revectoring" of the each of the calls from the single call site to each specialized function in turn – conceptually, turning the original call site into many call sites. Although it may be possible to create practical schemes to do this, the machinery and analysis involved is non-trivial, and it seems questionable whether the associated payoff is likely to be large enough to justify the associated complexity. See section 5.6.3 for further discussion of this issue.

[4]Note that it is not strictly true that the values at run time will be those expected by an analysis of the program text. Clearly a machine which had floating point operations performed in software could have an incompletely written value at a given point in the execution of the program, in this case, the "value" of a floating point quantity could be modeled as an operationally non-sensical but precise pattern of bits. Similarly, uninitialized variables may be non-sensical but precise.

The discussion above considered the character of the environment specified by stopping run time program execution at a particular point in time. Frequently during program analysis, however, we are interested in the values associated with program quantities at another form of point: we are interested in the program's states for a particular *point in the program code* during run time execution (assuming some *particular program input*). In some cases (such as the sole exit point for a program), the program execution may only reach that point in the code at a single point in time. In other cases, however, a given point in the code for the program may be executed several times in the course of program run time execution. In principle, we could collect and completely characterize the set of run time states of the program when it reaches this particular point in the code. Since each of these states is simply the state of a program at a given point in time, this collection of states would thus allow us to find the set of all possible values associated with any given program quantity at that program point for a given program input. Suppose now that we examine the set of possible values for a given program quantity within this set of possible states at this point in the code of the program. In some cases, this may be a single value – in this case, this program quantity always the same value every time we reach this point in the program (for the given program input). Other program quantities may be associated with several different values corresponding to different instants in time at which the program executed this point in the code. (For example, the point in the program code being examined may be within a loop, and the program quantity being examined may be a a loop induction variable, which is progressively assigned to values from 0 to 63.) Thus, at run time a given program may actually hold *many different values at a single point in the program code* – even considering execution only for a *fixed* program input. In such cases, it is clearly not realistic to hope to statically determine a single value for this quantity at this point in the program – for it is associated with *many* possible values.

Now consider adding an additional complication to our model of the program: Let us consider quantifying over all possible input values. While previously we were considering the states of the program for a *given pattern of user input*, let us now consider them for all possible input values. It should be stressed that conceptually, considering the values in a program over all possible input values is extremely similar to considering the values of function that is invoked from many different points in the program with different input values. Just as a value of a program quantity in a function may be known precisely for a *given input state* but be associated with different values for different input states, so a given quantity in our program may be precisely known for a given set of user inputs (and thus be statically known *with respect to these inputs*), but be associated with a set of different possible values for all possible user inputs. Consider a point in our program which for all inputs we only reach at a single point during our execution. While for a given set of user input the associated state will be completely known, this state may vary considerably for different sets of user input. Now if we consider the ensemble of possible user inputs, a given point in the code of the program will be associated with a set of possible states. Each of these states is in turn associated with some particular set of input values that leads to its realization, as well as with some particular instant in the evaluation of the program with those program inputs. Thus, while a given point in the code of a program is associated with some precise state *for every time it is executed for a given set of input values*, when we discuss the complete set of states associated with that point, we must consider the complete an ensemble of sets associated with the point *for every set of program inputs* **and** *for every instance of evaluation that reaches the program point for that set of inputs.*

This double quantification underscores the limits on what can be learned about the values of program quantities.

3.2.2 Epistemic Constraints on Static Knowledge

Up to this point we have been discussing a “run time view of program states” – the information that *does* actually obtain at run time, and the information that could in principle be learned about program states if we could actually watch the execution of a program proceed (for all given inputs). The sections above sketched the two inherent limitations on knowledge about the values associated with program quantities at a certain point in the source code: The facts that a given point in the source code can be associated with multiple run time states, and that one must quantify the analysis over all possible program inputs. Keeping these ideas in mind, we turn now to briefly examine the character of static program analysis. Program analysis is clearly not free to actually run the program on all possible inputs and to collect the appropriate state information for each program point – not only would such an option be prohibitively expensive, but it would would also void the need for program analysis, since the program would already have been completely run in all possible input configurations! Instead, program analysis is limited to simulation of the program with some limited knowledge of program inputs. (In existing compilers, no knowledge of program inputs is permitted at all; program specializers such as PARTICLE that implement partial evaluation can be given specifications of program input.) Given such incomplete knowledge, program symbolic evaluation must proceed by approximating each value as precisely as possible given the known constraints, the limitations on the size of the abstract domains, and the need to guarantee termination of the specializer.

It is important to note that for practical systems there can be no confluence guarantee here: The legal specializations resulting from a simulation of a program where the input value is known to be a member of some set (possibly just the universal set implicit in the value representation of “totally unknown”) is typically not the same as the possibly infinite intersection of the legal specializations resulting from the separate simulations of the program when the input is known to be each value of the set in turn. This lack of confluence derives from many different compromises necessary for the safe and efficient operation of the program specializer. A few of these compromises are discussed below.

Like other program specializers, PARTICLE makes use of abstract domains of finite height (although PARTICLE allows the height of the all abstraction, generalization, and computation lattices to be specified by the user, the given heights are always finite 3.3.4.). As a result, computations involving domain elements can result in other domain elements that must be approximated as “entirely unknown.” (e.g., In PARTICLE, the repeated addition of values represented by sets will yield sets of increasingly large size, until they are no longer representable as sets. Similarly, ranges will grow in size until the maintainance of the range information is deemed no longer worthwhile, and it is approximated by an “unknown” value). This approximation can prevent the recognition of certain opportunities for specialization that would be evident during the separate interpretations of the program with respect to each possible input value in turn (simulations which would involve only exactly known values, and require no approximations not imposed by the user for the sake of constraining running time or guaranteeing termination).

As another common and very important shortcoming of even very rich representations (such as those employed by PARTICLE), consider the program fragment in figure 3-1. The difficulty here is that although representation of the values of quantities as sets of possible values allows capturing of the value of each possible value *in isolation*, it does not expression *logical links between the values of different program quantities.* In particular, it may be that although program quantities a and b are each constrained at a certain program point to be the members of two different sets of values (for all possible instances of some partially specified program input), whenever a is a particular member of its set, b is some corresponding particular value of its' set. In other words, there may be some "covariance" between the values of a and b; any representation of each value in isolation will fail to capture this covariance.

Lack of confluence can also arise from approximations made in the simulation of control flow. In particular, PARTICLE joins other symbolic evaluation systems in enforcing "joins" in uncertain paths program control flow as close as possible to the associated "forks." For example, upon encountering an "if" statement associated with a predicate evaluating to an unknown value, PARTICLE is forced to create a conservative approximation to program behavior that applies no matter which branch of the *if* would actually be taken. (For an explanation of this behavior, see section 4.3.3.2.1.2.). In general, however, there are *many* legal approximations to a given program state, and many different ways in which to create such approximations PARTICLE enforces a join point at the end of the *if*, so that each branch of the *if* statement is separately simulated, a legal approximation to *both* of the resulting states is created, and execution of the rest of the program continues from this point. An equally valid method of creating a conservative model to program behavior would also involve the separate simulation of the effects of taking each branch – but instead of creating a generalized state immediately after the construct, one could continue separate symbolic evaluation of each branch for the remainder of the program. While this technique would be much more expensive than the method used by PARTICLE (yielding a number of distinct threads that is geometric in the number of unknown conditionals evaluated during symbolic evaluation),it would also prevent loss of covariance information for quantities assigned in the loop, allow the maintainance of much smaller approximations to program values (because of the need for fewer generalizations), and thereby allow more precise approximations to program values. Just as the system loses information about program behavior by failing to individually symbolically evaluate the program with respect each possible program input, so information is lost by failing to individually evaluate each possible thread of control within the program in isolation.

As a final example of the lack of confluence due to control flow approximations, consider "power" constructs such as loops and recursion – all of which allow arbitrarily many reexecutions of a given section of code. Intuitively, if we were to independently and exhaustively evaluate the program being specialized on each possible input value, the evaluation of the conditions for loop termination would always be known, and we would always know when to terminate a given loop. On the other hand, if we are attempting to simultaneously evaluate a program over a range of possible values of input values, some loop termination conditions may not be known. In some of such cases (see the next chapter for a more precise overview of these conditions), we are forced to conservatively approximate the iteration of the loop *arbitrarily many times* – a procedure that can lead to the adoption of a grossly conservative approximation to the effects of executing the loop. (The fundamental reason that the resulting approximation will be seriously suboptimal is discussed in the next chapter, in section 4.3.3.2.2.3; intuitively, it is simply because the consequences of all iterations become "mashed together" in the attempt to reach a fixed point.) Moreover, in order to generalize the effects of arbitrarily many iterations

```
if (u == 0)  /*   u   unknown   */
        {
        a = 2;
        b = -2;
        }
else
        {
        a = 1;
        b = -1;
        }

/*  after the join point here,
        a is known here to be element of {1,2}
        b is known here to be element of {-1,-2}
*/



/*  while total ALWAYS equals 0, PARTICLE fails to recognize that
    a is always 2 when b is -2 and that a is always 1 when a
    is  -1.  As a result, PARTICLE calculates the result of a+b as
    either -1,0, or 1
*/

total = a + b;
```

Figure 3-1: Although systems with very rich abstract domains may be able to precisely bound the possible run time values associated with program quantities, typically they model the value of each quantity in isolation, and therefore fail to capture the logical links between the values of these quantities. In the example above, whichever branch of the *if* statement is taken, the quantity $a + b$ will be 0. Unfortunately, because PARTICLE approximates the value of a and b independently, this invariant is not recognized.

within a finite amount of time, the height of the associated lattice for abstraction *must* be finite. (Note that no such constraint need apply to the height of the computation lattice, although a limit on the height of that lattice may be desirable for practical systems.) Repeated applications of least upper bounds to a program quantity will thus eventually yield values that are *entirely unknown* – even though it is possible that a slightly larger domain would allow an exact capturing of the set of possible values associated with that program quantity. The approximations necessary to enforce termination can thus lead to a dilution of static knowledge and prevent the realization of confluence.

As a result of these issues – practical compromises made concerning both the precision of representations and of the symbolic evaluation machinery – symbolic evaluation of a program given a set of possible input values frequently yields a model of the set of possible states at a program point for the given input values that falls considerably short of the ideal set of states that could be in principle be obtained by running the program separately on each possible input and collecting together the states at the program point for each separate execution (a set that is already possibly quite diverse). Nonetheless, we can still hope to achieve reasonable fidelity to such states, particularly if we have access to a set of rich abstract domains. As we will see below, in some cases program specialization itself can help us restructure a program in such a way that a program points will on average be associated with fewer diverse states. Such restructuring allows us to obtain more precise bounds on the run time values of program quantities, and will thereby permit a range of additional specializations to take place.

3.2.3 Approximations to Values

The section above provided an overview of the constraints and limitations on static knowledge. Symbolic evaluation is a high quality means of performing program analysis, and captures static knowledge concerning program quantities in the form of explicit approximations to program values and states. In light of the discussion on the limitations of static analysis presented above, it is worth briefly examining the character of these approximations.

Naively conceived, symbolic evaluation simply represents an interpretation of the program in which control flow is performed as accurately as possible given the information available, and computations operate on approximations to values. In such a system, each state at run time would have exactly one corresponding state approximation at compile time, and a given "value approximation" at compile time would represent an approximation to exactly one value at run time (the value resulting from the evaluation of some expression or residing in memory for some particular range of time).

In order to guarantee specializer termination (or just termination within some reasonable time), however, it is frequently desirable to simulate arbitrarily many iterations of program loops and levels of recursion within a finite time via the method of repeated abstraction. (See section 4.3.3.2.2.3.). Accomplishing this goal requires folding the effects of many run time iterations or recursions onto each simulated iteration of the loop or recursive function call. During this process of "abstraction", a given memory model maintained by the symbolic evaluator actually represents an approximation to many (and eventually *arbitrarily many*) run time states, and the "value approximations" maintained within this memory model represent approximations to the values associated with *many* run time states. Similarly, the evaluation of each expression within such contexts actually creates an approximation to the evaluation of expressions within *many* memory models. In such cases, it is not really legitimate to speak of each of the approximations manipulated at compile time as representing approximations to a *value* (i.e. as an approximation to a *particular bit pattern*) – instead, each "value approximation" is an approximation to a *set* of values (bit patterns) that obtain at run time. In this thesis, the term "generalized value approximation" will be applied to the concept of a value approximation quantified over all execution contexts that are approximated by the associated state; as will be seen in section 4.2.2, this concept plays a crucial role in the functioning of the symbolic evaluator.

The number of run time states approximated by a particular compile time memory model is a good measure of the coarseness of the symbolic evaluation being attempted. By setting the options to the symbolic evaluator so that loops are simulated in a more precise manner (see section 3.4.), there will be less use made of the mechanism of abstraction, the generalized values maintained within each simulated program state will need to approximate fewer distinct run time values, and will in general then be free to provide tighter approximations to the values being approximated than was previously possible.

This section has briefly examined the character of the compile time approximations to values and states. While finer-grained symbolic evaluation (evaluation that executes many loops and recursion on a iteration-by-iteration or recursion-by-recursion basis) allows each generalized value or memory model to approximate few (ideally, just one) distinct run time values or program states (and thus opens the possibility of maintaining more *precise* approximations), such fine-grained simulation carries with it considerable compile time overhead, and at some point termination concerns will force the adoption of more coarse-grained approximations.

3.2.4 The Effect of Program Specialization on Program Analysis

We turn now to examine the possible effect of program specialization upon the situations discussed above. In particular, consider again a point in the body of a loop with an induction variable that runs from 0 to 63, and a point in the code of a function with a parameter that is used but not written to (so that the parameter stays the same value throughout a given execution of the function, but may be associated with different values for *different* executions of the function (so that the parameter is static for a given call, but dynamic relative to the program as a whole). As noted above, even for a specific program input, the fact that the code in each case is executed many times with different states forces us to assign a set of possible values with the two program quantities of interest. In this case, even perfect program analysis (accomplished by executing the program for a particular input and watching the values adopted by the quantities of interest) would not allow us to obtain a precise value for the quantities. In short, the values of these quantities are truly dynamic in character. Now consider completely unrolling the loop discussed, and sneakily creating a specialized set of copies of the function so that for a particular specialized version of the function there is only a single value associated with the parameter value during run time (in other words, for all calls for a given specialized version of that function, the value of the parameter is the same). In this case, if we were to collect the set of all possible states of the program (for a given input) when it reaches a given point inside the loop, or a particular point inside a specialized function, we would find that they all were associated with precisely the same values for the quantities of interest. In these cases, judicious program specialization has permitted values that were truly (and not just perceived as) dynamic with respect to the remainder of the program to become completely static: *By spatially "spreading out" the temporally changing component of the computation, a program specializer can make static quantities that were inherently dynamic.* Program specialization allows us to create separate code (e.g., loop or function bodies) for different executions of instances of a given piece of code, and can thus enable more precise knowledge of the values of program quantities for each piece of code (thereby creating the opportunities for additional specialization).

Just as we created specialized versions of a function for different calls with shared characteristics in order to minimize constraints on knowledge arising from execution of the same code under significantly different states, we can create specialized instances of the program for different sets of *program* inputs to minimize constraints on knowledge arising from the execution of the program under significantly different sets of inputs.[5] As noted in section 3.1.3.5, by relaxing the conservative assumptions that otherwise must be made about about program inputs, such specialization can can yield substantial performance enhancements.

The discussion above has briefly discussed the interaction of program specialization and program analysis. Not only is specialization capable of directly exploiting patterns of data use to increase program performance (e.g., through mechanisms such as constant folding, collapsing of conditionals), but it can also play an *extremely* valuable role in loosening the restrictions on further specializations. Section 3.2.1 characterized the two *inherent* constraints limiting the precision of static analysis: The lack of knowledge about program input, and the association of multiple execution contexts with a single program point. The use of program specialization systems based on partial evaluation can eliminate the restrictions that traditionally arise from both of these fundamental constraints: By specializing a program with respect to some specified class of input, a program specializer can take advantage of knowledge of program input and eliminate the need to make overly conservative assumptions about that input. By creating copies of code separately specialized to sets of program contexts sharing certain patterns of data usage, the program specializer can greatly reduce the constraints imposed by the need to safely specialize code with respect to all execution contexts. The use of such can enormously facilitate the aggressive exploitation of further specializations.

3.3 Extensible Abstract Domains

3.3.1 Introduction

This section briefly surveys the costs associated with the maintainance of abstract domains of varying richness, and proposes the use of *extensible* abstract domains as a means of adaptively adjusting domain richness based upon the characteristics of the code being evaluated and the compile time/run time tradeoffs desired by the user.

3.3.2 Limitations of Restricted Abstract Domains

3.3.2.1 Direct Limitations

As seen in the last chapter, most previous work in the area of program specialization made use of rather limited representations for capturing the specializer's knowledge of program data. While such representations can efficiently capture of an important subset of knowledge about data, in some instances they are incapable of representing enough information to directly allow effective specialization to be performed. For example, consider a case where it is statically determinable that a given program quantity x is either associated with the value 0 or 1. While knowledge of such information would not allow us to perform constant constant propagation or folding, it might allow us to demonstrate that a given predicate (such as $x < 0$ or $x == 10$) is

[5]Conceptually, such specialization makes use of the partial evaluation capabilities of the program specializer, although in practice it is little different than specialization with respect to other internal values.

true or false, and thus to collapse a conditional and associated check (and implicit branch), or allow unrolling of a loop.[6]

The ability to maintain rich enough representations to permit elimination of such run time checks can be very important in certain circumstances. Section 3.1.3.4 discussed the performance advantages arising from the ability to eliminate run time domain checks. Exploiting many of these opportunities for specialization requires the maintainance of rich abstract domains. For example, in many cases the elimination of arrays bounds checks requires the ability to represent the fact that an index is within some known range of values.[7] The elimination of range checks within a graphics clipping routine or within a square root extraction routine requires a similar ability. As a final example, frequently it may be demonstrable that a pointer is *not* NULL (e.g., if it has just been allocated, or assigned to one of two pointers with exactly known referents), even though no particular value may be known for it. The ability to represent this form of information requires more sophisticated domains than are available in existing compilers or most in previous experiments in program specialization.

As mentioned in section 3.1.3.4, these domain checks can impose significant overhead, particularly in contexts where they are executed frequently (e.g., in a tight inner loop) or where the operation to be performed following the check is very simple (e.g. array lookup). In addition to eliminating the cost of the check itself, eliminating such checks enlarges the average basic block size, saves space and minimizes the deleterious effects of branches upon the pipeline. In many circumstances, the maintainance of rich domains is crucial for discovering and exploiting such specializations.

3.3.2.2 Multiplicative Effects of Ignorance

3.3.2.2.1 Introduction The section above discussed cases in which the inability to capture some knowledge about an imprecisely known program quantity can have *direct* effects in ruling out certain classes of specializations. In other cases, the inability to represent merely partial knowledge about a program quantity can have significant negative *indirect* effects. In particular, the failure to precisely bound the values possibly associated with a program quantity can lead to a "multiplicative effect" where such imprecision in knowledge results in an inability to capture precise about a range of other program quantities, or leads to directly to the loss of information about the values associated with many other run time quantities. In a qualitative manner, this phenomenon represents a program analysis version of Gresham's Law: "imprecise values drive out precise values." It remains to be studied how just drastic this effect is in common programs, but some examples will help to suggest the seriousness of the effect.

3.3.2.2.2 The Importance of Knowledge about Control Flow In the examples illustrating the direct limitations of restricted abstract domains, it was suggested that the main-

[6]The defining characteristic of expressions where imprecise knowledge is likely to be *directly* advantageous is the use of a many to one mapping of values to a new value. In such cases, while we may be uncertain about the exact values of the operands to the expression, it is possible that *all* of the possible values for the operands will yield the same result. In such situations, we can have *exact* knowledge of the result of the expression even when we have rather little knowledge about the operands to that expression.

[7]Note that such rich abstract domains are *not* necessary for cases where the code is iterating through the array *and* the associated loop can be simulated on an iteration-by-iteration basis. In cases where the loop must be abstracted (see section 4.3.3.2.2.3.), richer domains are required.

tainance of rich abstract domains can help eliminate a substantial category of run time checks and permit the collapsing of the associated conditionals. In some circumstances, substantial time can be directly saved by the elimination of such checks. Eliding these checks and conditionals can also have important indirect effects upon the quality of program analysis (thereby affecting the opportunities for discovering other optimizations). In particular, knowledge of the direction of evaluation within a conditional allows the symbolic evaluator to simulate the effects of the conditional by just evaluating the appropriate arm of that conditional. As will be detailed in section 4.3.3.2.1.2, simulating the result of a conditional associated with a predicate of unknown truth value requires independently simulating the effects of each arm of the conditional, and continuing execution from the end of the conditional using a generalized state that is a valid approximation to the state resulting from *either* branch. As a result, unless *both* arms of the conditional side effect a program quantity in exactly the same way, none of the side effects of the conditional on that quantity will be known definitively. For example, even if one of the arms of a conditional associated with an unknown predicate assigns all program quantities to precisely known values, a program specializer will likely be unable to take advantage of any sort of constant folding or propagation simply because it is possible that the other branch (which presumably does *not* make the same assignments) *could* have been taken, and because it is therefore unsafe to rely on the values assigned within just the single branch. If, on the other hand, the predicate of the conditional was known to evaluate a value that would definitively lead to the execution of the arm of the conditional involving the known assignments, enormously more precise static information would be available and would likely permit the exploitation of powerful specializations in the subsequent code. Thus, while the use of rich abstract domains to eliminate run time checks is highly desirable on account of the performance benefits accruing directly from the elimination of these checks alone, it is also important in allowing much higher quality static analysis to be applied to the conditional associated with the check. The higher quality of this analysis may permit the discovery and exploitation of many other specializations.

3.3.2.2.3 The Proliferation of Ignorance: Normal Operations Most operations that accept more than a single value on output will output values in which the uncertainty is proportional to the *product* of the uncertainty in the input operands. For example, if we wish to add together two values, the first of which is approximated by m possible values, and the second of which is known to be one of n possible values, in general our result will be one of approximately mn possible values (each consisting of the addition of a particular possible value associated with the first operand with a particular possible value associated with the second operand.[8] In short, if our knowledge is incomplete about even one operand to some operation, our knowledge about the result of that operation will likely be equally incomplete. And if our knowledge about *several* operands to an operation is incomplete, our knowledge of the result of that operand will be *grossly* incomplete.

3.3.2.2.4 The Proliferation of Ignorance: Indirection For some operations a slight lack of knowledge about the operands can far more have serious results for the quality of knowledge of the effects of the operation. Such is typically the case with operations that

[8]In particular cases, many of these possible output values may be equivalent, leading to a smaller set of output values.

involve *indirection*. In a sense, indirection can be thought of as implementing an extremely effective hash function: For a small change in the operand (the index or pointer) value to an indirection function, we can get a vastly different output value. In such cases, the use of a rich representation can make an enormous difference in the quality of the symbolic evaluator's approximation to the result of the operation. An example of this phenomenon follows.

Consider dealing with a point in a program program at which we have an array the contents of which are completely known (perhaps they have been given to us as static program input, or have been initialized in a certain static configuration). In addition, suppose that although the value of the array index at this point in the program is not fully determinable, it is statically determinable that at a given point the array index is only possibly associated with some small subset of the array (e.g., the first half, the first or last element, etc.) Consider a write of a known value to the array entry associated with this index. Our capacity for representing static knowledge about the values associated with program quantities will determine how drastic the resulting loss of knowledge about program values will be.

In many of the representations used in symbolic evaluators that were discussed in the last chapter, a value was limited to being exactly known or entirely unknown; in some systems, a value could be classified as an instance of some large intermediate class (e.g. non-negative value). In such systems, while the contents of the array are exactly statically known and can thus be represented *exactly*, the static information about value of the index is not representable, and must be conservatively approximated by a more "general" value – either "non-negative" or "completely unknown". In either case, we are forced to conservatively model the subsequent write as potentially being a write into *any* element of the array. Thus, when the write occurs the symbolic evaluator is obliged to update the state of all lvalues that may be affected by the write. Unfortunately, for each of these lvalues, it is *uncertain* whether or not its contents will actually be overwritten. As a result, the symbolic evaluator cannot simply update each possibly overwritten array element with the known value being written to the array – instead, the run time value associated with each array element must be approximated using an approximation legal both if the write did take place to this element *and* if it did not. In the systems being discussed, the only approximation to both possible values of the array element following the write is grossly conservative (such as "non-negative" or "completely unknown"). As a result, this single array write almost completely destroys our original information about the array; at most, we may know that the elements have some general characteristic (like being "non-negative"); in most of the representations used in past specializers, our model of the array after the write would give us *no* information about the array's contents.

A system with richer domains can allow considerably more information to be preserved in this face of the array write by virtue of permitting a more graceful degradation of knowledge about program values. Consider, for example, a system that allowed the specification of a value as fully known, a member of a set of possible concrete values (up to some size), or "totally unknown"[9] In this case, we might be able to completely represent the possible values of the array index at this point in the program – knowledge that would allow us to confine the effects of the write to our knowledge of the possible values of elements in a small section of the array, and thus lose extremely little information. Even for those elements of the array that may be overwritten, however, the more precise character of our representation allows us to preserve

[9]Both PARTICLE and REDFUN-2 offer abstract domains that include these subdomains.

a considerable amount of information: While for the systems discussed above the only legal approximations to the possible value associated with an array element after the write preserved very little information. Access to a set-based abstract domain allows us to represent each possibly affected array element as a member of a 2-set consisting of the original and written values. Thus, such a write will lead to a model of the array in which we know that each element in some range of array elements is one of two possible values.

As can be seen from this example, in some cases the options for knowledge representation about about a single value can have a huge effect on our subsequent knowledge of other values. In the example above, the richness of the abstract domains made the difference between having to partially dilute the information about a small portion of the array and having to completely invalidate all of the static knowledge about the contents of the of the array. In the presence of general and unrestricted pointer mechanisms, this multiplicative effect on fine-grained knowledge of program state can be even more dramatic (since an inability to bound the possible referents of a pointer can lead to a need to dilute or even completely invalidate our knowledge about a large number of values that are possibly affected [8][26]).

3.3.2.2.5 Representation of Input Constraints A final motivation for the use of rich abstract domains stems from the desire to permit the user of the program specializer to specify fine-grained constraints on program input. As discussed in section 3.1.3.5, there is a broad category of programs in which program input can be naturally staged because the user typically has access to one portion of the input long before the remainder of the input is available. In such cases, the input specification to the program during program specialization will typically include a large amount of exactly known data. In other cases, however, the input to a program may not be subject to such a clean decomposition. and the user may not be able to specialize the program with respect to input data that is *known*. While in such cases it is not possible to exactly specify any subpart of the input, it may still be possible to *partially* specify pieces of the input. For example, the user of a scientific program may only be interested in certain ranges of input parameters. Specializing the original program with respect to input data specified to be within these ranges may permit elimination of certain domain checks and handling for certain boundary cases, or even allow the eliding of algorithms designed for and only applicable to other particular ranges of input.

As argued in section 3.1.3.5, substantial performance advantages can be offered by even small amounts of knowledge concerning program input. In order to take maximal advantage of such knowledge, it is desirable to provide the user with a fine-grained means of specifying on program inputs. Values arising from program input are modeled by PARTICLE in exactly the same as those arising from values purely internal to the program. As a result, if the capacity for fine-grained specification is offered for program inputs, it would not only be natural to extend such the capacity for such specification to general values as well, but would be awkward and somewhat complex to avoid doing so. The desire to take advantage of the performance benefits allowed by exploitation of fine-grained input specification thus represents an additional motivation for maintaining general-purpose rich abstract domains.

3.3.2.2.6 Benefits of Rich Domains: Conclusions The sections above have sketched the direct opportunities for specialization resulting from the maintainance of rich domains and the importance of access to rich domains in slowing the effects of a form of Gresham's

Law for program analysis, and preventing the drastic loss of static information that can result from the lack of knowledge about control flow and writes to memory. Moreover, for program specializers performing partial evaluation (such as PARTICLE), sophisticated representations can allow precise partial specification of user program input, permitting the user to express some constraints for input parameters without leaving them entirely unknown or committing to particular values. As argued in section 3.1.3.5, even a small amount of knowledge about program inputs can frequently have significant effects on knowledge of values throughout to the program. As a result, the ability to express fine degrees of knowledge about program input values may yield important advantages during specialization. Having briefly surveyed the benefits of making use of rich domains, discussion now turns to analyze the considerable compile time cost of maintaining and manipulating values drawn from these domains during symbolic evaluation.

3.3.3 Cost of Rich Domains

While highly precise abstract domains can offer substantial benefits in the quality of program analysis and specialization, their use can be associated with high compile time costs. These costs stem from two primary sources:

- Rich internal representations require more space to represent and more complex and time consuming machinery to manipulate than simpler representations. As an example, consider the set-based representation of possible static values discussed above. In this case, a program quantity is approximated by a set of possible values it that can adopt at run time. Because evaluation in the abstract domain must simulate the effect of each operation upon elements in the abstract domain, the maintainance of abstract domains of this complexity requires a correspondingly more involved abstract evaluation mechanism. Consider, for example, the abstract interpretation of the addition of two program quantities. Suppose that the value associated with each of these quantities is approximated by a set of possible run time values it can adopt; suppose that the first value is associated with m possible values, and the second with n possible values. Assuming that we do not abandon the set approximation for the result of the operation for some sparser option (such as as the approximation "unknown", or a range representation in PARTICLE), performing the addition requires the creation of a set representing all of the possible values resulting from the addition. Since each of the operands is considered as possibly associated with each of the values in its associated set, the value resulting from the addition could be the result of adding *any* value from the first set with *any* value from the second set. As a result, the resulting set must be generated by pairwise addition of each element in each set – an operation involving mn operations.[10] Similarly, an equality test between two values approximated by a set of possible values requires $O(mn)$ operations. As a result, the overhead for simulation of operations in such rich domains can be substantially greater than that required for simple domains. (For example, in a domain consisting only of fully known values or entirely unknown values such as that commonly employed in constant

[10]As noted in section 3.3.2.2.3, some of the mn combinations of elements may yield equivalent results, but in general all mn operations must be performed.

propagation [2], the addition would at most consist of two classification checks possibly followed by a single addition of two domain elements)

- The second major source of costs associated with the maintainance of rich abstract domains arises from the need to simulate the effects of arbitrarily long computations within some finite time. The use of sophisticated abstract domains of permits us to distinguish between a greater number of possible patterns of run time values with which a quantity can be associated. As will be discussed in section 4.3.3.2.2.3, performing safe abstract interpretation on a loop in the absence of exact knowledge of program control flow requires creating a model of memory that is a valid approximation to the program state following the loop after *any* possible number of iterations. In order to obtain this state approximation, we repeatedly execute the body of the loop and after each iteration create a legal approximation to the program state after each of the loop iterations yet simulated. The symbolic evaluator continues this process until executing the loop another time and generalizing with preexisting results of the loop *has no effect* upon our approximation of the outcome of the loop – in short, until the loop could execute *arbitrarily many more times* without changing our approximation of the resulting state. At this point, we know that our approximation to the result of executing the loop is a valid approximation to the result of executing the loop for *any* number of iterations. Unfortunately, when we are capable of creating very flexible and extensible approximations to the values associated with a program quantity, the iteration process can take considerably longer to converge (simply because we allow more representations for the possible values a variable could be associated with the end of the loop before we reach a sufficiently general one to serve as a conservative approximation to the value for *all* possible iterations of the loop.) Thus, while richer representations may allow much more precise knowledge of the effects of a loop than do simple value representations, the compile times needed to approximate the effects of loops whose execution is predicated on unknown loop conditions (or whose length execution exceeds some user-dictated limit) can be substantially longer than those needed for simpler domains.

3.3.4 A Flexible Middle Path: Extensible Abstract Domains

The sections above have emphasized the significant benefits and substantial compile time costs that can result from the use of rich abstract domains. As a result of these benefits and costs, program analysis experiences a tension between the need for a rich representation to allow high quality program specialization, and the desire for a simple representation to allow rapid program analysis. Moreover, additional tensions may arise because different levels of precision may be desired for different categories of program quantities: For instance, it may be highly desirable to have program pointers represented as exactly as possible, while the possible values individual elements of a large array need not be captured as precisely[11] One manner in which to address the simultaneous desires for high quality analysis, low compile time cost, and varied needs for

[11]While aggressive compilers tend to make use extremely simple domains for most program analysis and specialization (e.g., constant folding and propagation [2][53]) and have no capacity for modeling the values associated with individual array elements, richer (e.g., set-based) aggressive domains may be employed to track the values associated with more sensitive program quantities such as pointers [2].

domain richness is to have the representation used during symbolic evaluation be as *flexible* as possible – to permit the use of rich representations of program quantities where desired, but to avoid requiring their use. In this way, the user can choose the particular tradeoff between precision and compile time that is most appropriate for their particular situation. For example, once program development has completed, a final highly-optimizing compilation process may be run to create a thoroughly specialized output program. For intermediate stages of program development, a less thoroughly analyzed and specialized program may be used for performance analysis and debugging. Within the symbolic evaluation of a given program, such flexibility can be equally important: Highly precise analysis can be used for areas of the program whose performance is particularly critical (e.g., extensive program loops or recursive sequences), or areas where less precise analysis is unlikely to be effective.

In keeping with its general philosophy of offering the user maximal control over the range of choices made during program specialization, the PARTICLE system makes use of a general approach allowing great flexibility in the richness of analysis performed during symbolic evaluation: It allows the use of arbitrarily large set sizes for capturing possible values of variables (where the user is permitted to specify the heights of the different value lattices associated with abstraction of program quantities and with simple computation.) Moreover, in observance of the importance of pointers in program analysis, PARTICLE permits separate specification of the domain richness to allow for pointers and non-pointer program quantities.[12] Moreover, if desired, the user can dynamically insert instructions into the code to be executed, directing the symbolic evaluator to allow more sophisticated or enforce simpler representations for values during subsequent execution. The mechanisms used permit the user to control the richness of compiler analysis during symbolic evaluation in a very fine grained and fully dynamic manner, depending on the particular tradeoffs between precision of analysis and compile time that are desired for particular pieces of code. (These directives are given dynamically to the symbolic evaluation system through the use of calls to appropriate external call stub routines: See section 4.3.5.1.2.)

3.4 Guaranteeing Specializer Termination

As noted in chapter 2, some of the most practical and high quality program specializers to date have adopted a straightforward but somewhat loose policy towards guaranteeing termination: In such systems, program specialization is guaranteed to terminate if the program being specialized is *known* to terminate [56]. In other situations, no guarantee is given as to specializer behavior. For certain ranges of programs, such a policy is not unreasonable: In many instances, a program performs some specific computation and then terminates. Unfortunately, there are also certain classes of programs that do not fit into this pattern. For example, operating systems are designed to run continuously, and may be associated with no guarantee of termination. Consider

[12]It would be desirable to allow the user to separately specify the richness of domains to associate with other classes of quantities as well, such as array indices. Ideally, separate specifications of the domain richness would be permitted (although not enforced) for individual program quantities, and could be adaptively adjusted by the specializer system based on feedback during symbolic evaluation. Similarly, it would be desirable if PARTICLE would sense the desirability or needs for more precise program analysis within certain sections of the program (e.g., program loops) and adaptively adjust the richness of the representations employed to reflect these needs. See section 5.3.2.1.

also certain classes of programs performing successive refinements of calculations (e.g., programs calculating the digits of π or successively approximating certain physical systems to increasing degrees of precision) – such programs may simply be designed to run forever. Providing simple guarantees of evaluator termination under *all* conditions requires little machinery that is not already included in the symbolic evaluation system, it seems natural to provide the user the option of requiring program termination under all conditions.[13]

The manner in which general termination is enforced in PARTICLE is very simple: User-specified or default limits are imposed upon the number of successive recursive function calls or loop iterations that are actually simulated before a process begins to create an approximation to the effects of arbitrarily many additional recursions or iterations within a finite amount of time. PARTICLE recognizes two major subdivisions of such recursive calls or iterations: Recursive calls or loop iterations in which the recursion or loop iteration is *known* to actually take place at run time (given that the first function invocation or iteration did), and those circumstances where it is *not* known whether the subsequent recursion or iteration will be executed (where there is some sort of conditional associated with a predicate of unknown truth value separating the two function invocations or the two iterations of the loop.

For loops associated with known iteration predicates or recursive function calls where the recursion is known to take place, the user may be *far* more willing to execute the recursive calls or loop iterations, precisely *because* it is known that if execution reaches the construct being evaluated (the loop or original call site), all of the computations being simulated *will actually take place at run time* – rather than simply spinning its wheels and speculatively executing code whose execution at run time is merely *possible*.[14] Moreover, some of the most important optimizations possible during program specialization involve and require the lengthy and full-bodied execution of loops and recursion, and thereby serve to shift substantial amounts of computation from run time to compile time. As discussed in section 2.4.2.2.2, the type of rich symbolic evaluation that is necessary to take full advantage of static information *requires* the ability to perform full-bodied simulation of constructs such as loops and function calls. In such conditions, a user may wish rather to apply rather loose throttling strategies for symbolic evaluation. In particular, because the loop or recursion may well be statically performing some significant computation and thereby eliminating it from run time, it will likely be perceived as desirable to allow the symbolic evaluator significant freedom in executing the loop.

[13]It should be stressed that the simple strategy adopted to enforce specializer termination in PARTICLE is very simple in nature and could very easily have been integrated into past specializers – the enforcement of program termination in all circumstances does *not* represent or reflect a technical advance over such systems.

[14]The current discussion is somewhat sloppy in its willingness to apply the term "speculative" for the symbolic evaluation of code within a loop whose termination condition evaluates to an unknown value, and to speak of code in loops with known recursion or iteration conditions as "non-speculative." In fact, the entire loop or function call may itself occur within a speculative context (e.g., inside a conditional – or many levels of conditionals – whose predicate evaluates to an unknown value). By referring only to the quality of static knowledge concerning the truth value associated with the loop or recursion condition, PARTICLE makes use of a a very simple, local criterion to select the desirable strictness of the limits on iteration and recursion. This criteria provides a first-order hint to determining how much effort the symbolic evaluator should put into precise evaluation of the associated construct, but is grossly simplistic. More sophisticated strategies would make use of information concerning the *overall* likelihood that the recursion and iteration would take place (information indicated by global context – a function both of the total nesting of speculation encompassing the statements and the number of times the context surrounding the current code will be executed at run time), and feedback concerning the amount of *useful work* that such iteration is actually accomplishing. (See section 5.3.2.1.2.).

In many instances, the iteration or recursion condition associated with a loop or function may not be associated with a known truth value. In such cases, it not known whether the iteration or recursion will actually take place at run time, and the user may wish to avoid spending significant computational resources precisely evaluating the effects of speculative code. Moreover, once an iteration or recursion condition is unknown, it is likely to remain unknown for future iterations or recursions[15], and it is unlikely to be beneficial to continue precise simulation of each recursive application of the function or of each loop in the hope that the situation will change. Moreover, because each iteration or recursive step is only *possibly* executed, even the most precise simulation of the effects of each such step will yield inexact knowledge of program values following the execution of the loop or root function call, due to uncertainties about control flow (see section 3.3.2.2.2.).

In keeping with its general philosophy of permitting user maximal control over the the character of the specialization process, PARTICLE allows the user to separately specify the running time constraints enforced for each of the classes of situations above, thus permitting a continuous tradeoff between very long compilation times permitting allowing very high quality (but perhaps wasteful) simulation of the details of program control flow and modes which sacrifice this precision for faster compilation times. Moreover, by permitting the user to set the constraints associated with each form of looping via calls to the external system during symbolic evaluation, PARTICLE allows very fine-grained (and dynamic) specification of the constraints to impose on looping and recursion. While specializer termination is guaranteed under all conditions, this capacity offers the user the *option* of tailoring the quality and speed of program analysis to the details of the program being analyzed and the compile time/run time tradeoffs desired by the user.

Although the mechanisms used within PARTICLE allow finer-grained specification of loop simulation parameters than existing symbolic evaluation or compiler systems, they share the fundamental pitfalls of such systems (e.g., [56]) In particular, while the maintainance of maximal bounds on loop iteration may represent a reasonable (or at least not grossly inappropriate) mechanism for effecting the desire for more rapid or precise analysis of loops, it is seriously inadequate as an automatic throttling mechanism for intelligent balancing tradeoffs between high quality analysis and analysis time in loops with *known* iteration conditions. In particular, while the enforcement of a maximum bound on the number of known iterations to be executed has the effect of ruling out the need to execute the full loop in the cases where this would be least desirable (in instances of extremely long loops computing little new static information or loops that do not terminate), it also has the effect of ruling out the possibility of performing exact of loop analysis in cases where it can be most beneficial. As will been seen in section 4.3.3.2.2.3, the approximations to output state gained by performing iteration-by-iteration simulation of loops is typically *much* more accurate than that obtained by performing abstraction upon the loop body. For very long loops, it is likely that a great deal of data is being manipulated by the loop. Under certain conditions, iteration-by-iteration simulation of extended loops can allow vastly more precise modeling of the effects of the loop upon the data than is possible when abstraction is performed; in other cases, the loop may be computing little or no statically

[15]There are a number of different reasons for this tendency. One of the more important explanations is linked to Gresham's Law for program analysis (discussed in section 3.3.2.2.) – if even a small amount of uncertainty about program values is present at one point in the symbolic evaluation of the program, the uncertainty is likely to be magnified as evaluation continues.

known data and the approximations to program behavior gained through iteration-by-iteration emulation will offer virtually no advantages over those obtained through loop abstraction. Thus, the simple bounding on the number of permissible iterations to execute in a loop with known iteration conditions has the great advantage of preventing specializer non-termination and can prevent the needless and fruitless simulation of lengthy loops, but under other conditions can have substantial cost on the quality of program analysis permitted. (See figure 3-4.). The creation of *sophisticated* termination strategies for symbolic evaluation systems is very much an open research problem [22][56]. Ideas for strategies that enforce termination while facilitating high quality analysis and the better allocation of compile time are in section 5.3.2.1.2

It should be noted that under certain conditions the invocation of abstraction to shorten the time required for loop analysis can actually have the reverse effect. For example, consider the first case illustrated in figure 3-4: During loop abstraction, the indices to the array will become imprecisely known (eventually spanning most of the array), and each iteration prior to convergence during the loop abstraction process will involve the simulation of writes to *most* array locations. Because several iterations will likely be necessary prior to reaching a fixed point (the exact number will depend on the contents of the array and the richness of the domains employed), the abstraction of the loop may be very expensive. By contrast, the fine-grained simulation of the original loop will require only a single write to each array element and may require less compile time than the abstraction process.[16].

3.5 Issues in the Choice of an Input Language

3.5.1 Introduction

Chapter 1 noted that relatively little work has been devoted towards the creation of automatic program specializers for imperative and low-level languages, despite the enormous popularity of such languages in software development. PARTICLE is designed specifically to explore the potential for specialization of such languages, and the language C [31] was selected for processing by the system as a particularly popular and challenging representative of this class of languages. This section examines some of the motivations for this choice, and briefly describes the desugaring system used as a preprocessor by PARTICLE to minimize the tedium associated with the processing of the full C language.

3.5.2 Inherent Advantages of C

C offers a number of advantages as a source language for experimenting with program specialization of imperative and low-level languages. Some of the more important benefits of this choice are discussed below.

3.5.2.1 Generality

The C language permits the user great freedom in the manipulation of pointers, types, and heap structures [40]. In certain cases, such freedom can greatly hamper precise static analysis and

[16]On the other hand, if the interpretive overhead associated with statement parsing is considerable, such costs are likely to add considerably to the compile time cost of the fine-grained loop simulation within larger loops.

```
/*  all elements of array known */

for (i = 0; i < ARRAY_SIZE - 2; i++)
        {
        array[i] = array[i + 1] + array[i + 2]
        }
```

Figure 3-2: This figure shows a code fragment in which a simple static array calculation can take place at compile time. During loop abstraction, however, the exact values of the index variables are lost during iteration towards a fixed point, it becomes impossible to model loop effects with any accuracy, and in most circumstances the symbolic evaluator will lose all or virtually all knowledge of array contents.

```
int BinaryCompare(char *sz1, char *sz2)
  {
  for(; *sz1 == *sz2 ;  sz1++, sz2++)
    {
    if (*sz1 == '\0')
        {
        return(0);
        }
    }
  return(1);
  }
```

Figure 3-3: The code fragment shows the code for a string compare routine that returns 0 when two strings are equal, and 1 otherwise. Fine-grained simulation of the loop permits exact derivation of control flow and calculation of return values and allows collapse of the loop structure and internal conditionals. During loop abstraction, the symbolic evaluator loses track of the positions of the cursors within the strings and is unable to derive any of these pieces of static information or perform any of the associated specializations.

```
/* n known to be 100 */

a = fact(n);
```

Figure 3-4: Some of the longest-running compile time calculations can be associated with computations that whose shifting from run time to compile time can yield substantial run time speedups. In cases such as that pictured above, it can be difficult to distinguish such computations from those whose simulation will offer little run time benefit. Abstracting over such the call shown will entirely eliminate the possibility of shifting the computation to run time and inhibit further specializations based on the static return value of the call.

severely limit the range of assumptions that can be made about program values. The creation for such an unrestricted language of a program specialization system working through symbolic evaluation would provide a constructive proof that virtually all low level and imperative languages could also be handled. Moreover, if a program specialization system such as PARTICLE were to offer substantial benefits for such a language, it seems likely that similar systems could offer equal or greater performance advantages in other, more constrained, languages.

3.5.2.2 Size of Language

The C language is small and simple enough to permit an individual to implement language processing tools within a reasonably short period of time. (Particularly when such tools need only themselves process the subset of C emitted by the desugaring mechanism discussed in section 3.5.4.) While a smaller language such as PASCAL would be simpler to analyze and manipulate, it would lack many of the important low-level challenges of C. On the other hand, an ability to handle the semantic complexity of imperative languages such as such as PL/I, MODULA-2 or ADA would require a significantly more extensive implementation effort. C represents a small but interesting compromise between these two classes of languages.

3.5.2.3 Challenging Issues

C offers a number of challenges to high quality program analysis. Many of these difficulties stem directly from the unrestricted and low-level character of C and represent interesting topics for research in the specialization of similar languages. A few such challenges are discussed in this subsection.

3.5.2.3.1 Pointers The importance of high quality pointer analysis in low-level languages has been widely recognized [54][16] [35][8]. As discussed in section 3.3.2.2.4, even slight amounts of incremental compile time knowledge about pointer values can prevent substantial losses of static information concerning program quantities. The combination of general-purpose pointer manipulation and completely unrestricted memory reads and writes makes analysis of pointer-intensive C code particularly difficult. In certain circumstances, the high quality modeling of pointers within programs written in C can eliminate the need to adopt grossly conservative assumptions concerning the effects of indirect writes and under certain conditions can substantially improve code performance.

It is worth making special mention of one class of pointers in C programs for which high quality analysis is particularly important: Pointers to functions. By effecting calls to referents that are not capable of being statically determined, pointers to functions can prevent precise static knowledge of the call graph and control flow of a program. Ignorance of patterns of program control flow can in turn have substantial impact on the data flow assumptions a compiler can safely maintain. The richer store of static information available during symbolic evaluation may help to bound the referent of a given function pointer to one of a small set of possible referents, allowing substantially better knowledge of the dataflow effects of calls made through such a pointer. In some cases, the quality of information available during symbolic evaluation may permit exact determination of a function pointer's referent thereby allow the conversion of indirect calls through that pointer to a direct invocation of the target function (or even allow the inlining of the referent function). As discussed in section 3.1.3.2, such optimizations can yield particularly important performance advantages in object-based environments associated with type-based method dispatch.

3.5.3 Popularity and Widespread Availability of C

While the popularity of a language might initially be dismissed as irrelevant when consideration its use in a research project, such popularity can carry with it a number of important secondary advantages. Some of the most significant consequences of C's popularity for the current project are briefly mentioned here.

3.5.3.1 Many Good Compilers

The popularity of C has encouraged the development of many high quality compilers for that language. The availability of these compilers offers several major benefits:

- Because existing compilers provide a convenient means of translating from C to the machine language level, PARTICLE can operate entirely as a source to source transformer and rely on a supplementary compiler to complete the translation of the residual code. Such a separation of labor considerably decreases the amount of work necessary to build the specialization system and allows PARTICLE to inherit the platform-independence of the C language. Moreover, by comparing the performance of compiled input source code against the performance of the compiled residual program, such a system allows the isolation and precise determination of the benefits and costs offered by PARTICLE's general strategy for program specialization independent from the benefits and costs associated with the remainder of the compiler.

- The availability of aggressive compilers for C permits many higher-level optimizations that might otherwise have to be built into the partial evaluator to be delegated to the compiler used to compile the residual code. Dividing the work in this manner sacrifices the opportunity to use of the superior quality of information gathered by the partial evaluator in lower-level optimizations but saves substantial implementation time.

- Finally, the existence of highly optimizing compilers for C allows for a comparison of the quality of analysis and relative performance benefits offered by traditional compiler techniques and those permitted by symbolic evaluation in areas where the functioning of such sets of systems overlap.

3.5.3.2 Large Number of Applications

The tremendous number and diversity of software systems written in C permits the performance advantages offered by PARTICLE to be surveyed for source programs of vastly varying character. While some of these systems may exhibit substantial performance improvements from program specialization others may yield far less dramatic results. A study studying the range and patterns of performance boosts offered by program specialization and the quality and character of program analysis needed to obtain these performance enhancements could contribute substantially to an understanding of the strengths and weaknesses of program specialization and compilation technology in general.

3.5.3.3 Widespread Availability of C

C is supported on a broad range of architectures and under a variety of operating environments. The diversity of this support permits the performance advantages and costs of program specialization to be compared in a wide range of contexts. Such experimentation may help to highlight universal benefits of program specialization and may also draw attention to cases in which the benefits of specialization are distinctly variable. For example, certain patterns of specializations may be undesirable when associated with certain ranges of of cache or primary memory sizes, and desirable in other contexts due to increased of basic block size. As an another example, it may be that by relaxing interprocedural data flow constraints the use of program specialization would prove particularly beneficial when paired with existing compilers for parallel architectures. The frequency and magnitude of these effects is of obvious interest in judging the practicality and applicability of partial evaluation for imperative languages in general.

3.5.4 Reducing Syntactic Complexity through Desugaring

While the static semantics of the C language are relatively simple, the language syntax is relatively complex and involves a great deal of duplicated functionality. A language processor for C has the tedious job of dealing with many distinct syntactic constructs having identical or nearly identical semantics. For example, C offers the users ten similar structured looping constructs, and there are frequently a variety of manners in which to phrase a given expression (e.g., $++x$, $x+=1$, $x=x+1$, $(x=x+1,x)$). In some cases, expressions and statements are themselves interchangeable: a conditional assignment can be encoded either using the "?:" operator or performed within a conditional statement. Such syntactic flexibility can permit terse coding but is rather tedious and uninteresting to handle fully.

In addition to requiring the monotonous processing of a variety of semantically similar constructs, complete handling of C requires the proper treatment of semantic details in areas such as implicit type casts, bit field manipulation, and the processing special case constructs of various sorts (e.g., string initializers).

Finally, the structure traditional C is somewhat awkward in light of the particular desires of program specialization and symbolic evaluation: It would be ideal for inlining if all function calls took place within the top level of their containing expression, and very convenient for the remainder of the system if all (non-call) expressions were guaranteed to be free of side effects.

Fortunately, the vast majority of the tedium and annoyances associated with the processing of C can be factored out of the program specialization system by creating an autonomous desugaring mechanism to translate ANSI C code into semantically equivalent but syntactically simpler Simple C code. By passing all input code through the desugaring mechanism prior to processing, the program specializer or any other language tool is then responsible only for the handling of the canonicalized and considerably simplified Simple C language. The desugaring process can take place with a minimum of performance cost and allows language tools to avoid the tedium involved in the handling of a wide range of uninteresting details. PARTICLE makes use of an autonomous desugaring mechanism that greatly lessens the range of constructs and semantic considerations that must be handled by the specialization system. Table 3.5.4 describes the performance cost (and occasional benefits!) associated with desugaring for a number of small programs discussed in chapter 6. Although two of the five examples given illustrate non-trivial performance changes due to desugaring, the other three cases show virtually no performance change.

Program Name	Original Runtime	Desugared Runtime	Desugaring Speedup Factor
Matrix Multiply	5.92	5.9	1.003
Poly Mandelbrot	63.4	63.7	.995
Bessel Fn Eval.	2.81	3.18	.884
Dhrystone	3.16	2.95	1.07
Linear Interpolation	11.37	11.35	1.002

Table 3.1: The table above describes the performance consequences of desugaring on a small suite of sample programs described in section 6.2. Column 3 describes the ratio of the run time of the original code to the run time of the desugared code. As can be seen from the table, desugaring is typically associated with rather modest effects on program performance, although there can be some performance variation in either direction.

3.6 Conclusion

This chapter has reviewed some of the primary motivations for the use of a compilation system performing high quality program specialization. The use of program specialization can offer dramatic performance benefits in a number of important cases, and by spatially “spreading out” the temporally changing components of computations and through the use of specialization with respect to program inputs program specialization can considerably lessen the inherent limitations on the quality of program analysis. The sections above discussed the advantages of basing such a specialization system upon a flexible symbolic evaluation framework making use of values drawn from rich and arbitrarily scalable abstract domains. Having motivated and sketched some of the most important design decisions made in the creation of PARTICLE, the following chapter sketches the manner in which the system is constructed and operates.

Chapter 4

System Impementation: Abstractions and Mechanisms

Chapter 3 examined some of the conceptual background to program specialization and symbolic evaluation, and discussed some of the high-level design decisions made in the design of PARTICLE: issues such as the representation of static information, the throttling of specialization, and the selection of optimizations performed. This chapter shifts the emphasis to describe how PARTICLE operates: In particular, it provides an overview of the manner in which PARTICLE performs its job, emphasizing the most important algorithms and data abstractions involved. Unfortunately, PARTICLE is a large project and many important aspects of the system must necessarily be omitted from the discussion altogether. The first section of this chapter serves to introduce the fundamental abstractions employed in the system and sketch the basics of their use. Later sections discuss the major algorithms employed in system operation.

4.1 Overview

PARTICLE is a program specializer operating by symbolic evaluation. Particle accepts as input an input program written in a low-level imperative language, and outputs a transformed version of the input program in the same language. The fundamental structure of the system resembles that of a language interpreter. The system's evaluation both conceptually and in practice consists of abstractly interpreting the source code of the given program.

4.2 Fundamental Abstractions

4.2.1 Abstract Syntax Representations

Before we can symbolically evaluate a program, we must have some manner of representing it. PARTICLE makes use of a tree-structured abstract syntax representation for the constructs of the input program. Because PARTICLE operates by source-to-source transformation of its input, such representations remain resident throughout the symbolic evaluation of the program, and the new code created at the end of the evaluation process also makes use of such data structures.

PARTICLE is written in an object-oriented fashion, and the abstract syntax trees are associated with not only the program representation itself, but also with the means to transform it. In some sense, although they are in some sense "static"' entities, the elements of the abstract syntax representation of the program in represent the fundamental framework upon which the process of evaluation is performed. It is the algorithms associated with these representations that perform the symbolic evaluation of the program, and that make use of the more dynamic abstractions discussed below. Expressions are responsible for evaluating themselves into values in the context of a globally established model of program state (note that expressions are entirely side-effect free within the language processed by PARTICLE; see section 3.5.4.); statements map an input program state to some output state. In each case, the appropriate expression or statement will report any opportunities for possible specializations discovered during its evaluation. In particular, in addition to an object of its normal return type (state or value), the evaluation of the construct will yield a "reduction template" that specifies (and will perform during code generation) any specialization detected during construct evaluation and can estimated time/space savings and costs. If it is compatible with previously recorded optimizations for this construct during the current call, this reduction template is associated with the appropriate construct and with database of reductions for the executing call to the currently active function.

Some abstract syntax constructs maintain additional information related to their evaluation. For instance, loop statements maintain information relating to their degree of unrolling, and labeled statements that can be targets of gotos must record generalizations of the memory models associated with all paths of execution that have reached them but not yet been executed.

4.2.2 Values

While the values manipulated by expressions in standard interpreters are always concrete (always fully known and precisely specifiable), a symbolic evaluator such as PARTICLE must deal with uncertainty concerning the precise values associated with program resources. Within standard language interpreters, a value represents a particular piece of data (pattern of bits) that can result from the evaluation of a particular expression at run time or can be stored and retrieved in some program resource (e.g., memory location or stack pointer). The existence of a value in the computation of a program can be highly very brief (e.g., if it is merely a result of evaluating a subexpression), or can be prolonged (e.g., if it is stored in memory).

As discussed in section 3.2.3, each memory model approximation maintained by PARTICLE may actually simultaneously represent an approximation to *many* run time states. Similarly, a given "value approximation" within PARTICLE may represent an approximation to *many* run time values. PARTICLE makes use of the notion of a "generalized value approximation" – an approximation to a value that is quantified over all contexts being simulated by the associated memory model. For example, the value could represent the result of some expression for the many different iterations of some loop that are being simultaneously simulated by the current path of execution; while for a *particular* iteration of that loop at run time, the result of that expression will be some particular value, since we are simultaneously simulating many different iterations of the loop, in general we cannot speak of the expression as having some particular *value* (or bit pattern) – we must instead speak of the *generalized value* of the expression. The concept of a generalized value approximation is embodied by the Value abstraction.[1] Values play a central role in the operation of the symbolic evaluator, and are fundamental to the operation of the system.

PARTICLE makes use of Values drawn from its abstract domain. Each expression operator encountered during the operation of the program is performed to as high a fidelity as possible using these abstract values (which may sometimes directly represent *concrete* values), and produces a new abstract value. In the process of symbolic evaluation, we can hypothetically[2] *learn more* about the possible set of run time bit patterns associated with a Value, but the Value itself (like the values it approximates) is an immutable object and represents some *particular* set of bit patterns. While a Value may reside in some particular location in memory or in some other program resource and can be replaced (or partially replaced) by another Value placed in that resource, the Value itself never changes. Thus, a Value is an immutable, untyped quantity of some particular size that approximates some set of run time values. For reasons discussed in section 3.2.1, a Value of a program quantity may be less than exactly known, either because of epistemic limitations of the symbolic evaluator or because of inherent limitations imposed by program structure and/or lack of knowledge of program input.

Each Value manipulated by the symbolic evaluator is associated with several pieces of information. This section briefly overviews each subcomponent of this information.

4.2.2.1 Identity Tags

Each Value is associated with an (implicit) "identity tag" that distinguishes it from all other Values not known to have *exactly* the same run time value (for all contexts). Intuitively, the identity of the Value allows the notion of "known sameness" to be applied to Values. Note that while it is true that if two Values have the same identity, then they represent the same run time value, it is *not* true that different Value identities imply different run time value – it simply means that they are not *known* to be the same value in all contexts. Using this metric of identity, we can perform effective equality tests even in the presence of uncertainty regarding the precise bit patterns that will associated with a Value: While the particular bit patterns associated with two particular Values may not be known at a certain place in the code, we can test if the Values are *known* to be the same by comparing their identities. Without a mechanism such as this, is no effective manner exists of establishing equality of Values that are not precisely known. Consider figure 4-1: Here the Value associated with local variable x is completely unknown. Variable y is assigned x's value and later compared against it. Without the maintainance of some sort of mechanism for establishing identity, we would be forced to regard x and y as *possibly* in the accordance with the specified relation – we have no way of establishing the logical links between their possible Values and showing that the relation cannot

[1]This section adopts the convention of making all references to the Value abstraction that captures the concept of a "generalized value approximation" using a capital "V". The lower case "v" is reserved for references to the notion of a "value" (a *particular bit pattern*). For a review of the difference between these concepts, see section 3.2.1.

[2]PARTICLE does not currently offer any mechanism to allow the symbolic evaluator to gain additional information about values following their creation. REDFUN-2 [25] offered such a capacity in the form of "conditional contexts" (where the predicate associated with a conditional would be known to be true when evaluating the consequent, and false when evaluating the alternative), and support for this powerful means of collecting static information may be included in future modifications of PARTICLE. See section 5.5.3.

possibly true.

In addition to maintaining Value identity for direct transfers of Values (such as occurs in Value assignment), PARTICLE allows the user to enable or inhibit the general application of this principle, where preexisting Value identities are preserved under recalculation. When this option is enabled, the system will yield a Value known to be exactly identical to previous results whenever it detects the reapplication of a previous operation to the same ordered set of operands. As an example of this, suppose that program quantities a and b are be completely unknown, and consider the two instances of the expression $a + b$ at different points in the code, where the code contains no writes to a or b between the two instances of the expression. Because a and b are are unknown, that the result of the operation is a Value whose corresponding run time bit patterns are completely unknown. If the complete mechanism for enforcing Value identity is in place, while each evaluation of the expression will yield unknown values, the two values produced in this manner will be given the same *identity*. Thus, while we will know nothing of the actual run time bit patterns associated with the results of this operation, we will be able to establish that the values produced by the two different subexpressions are always equivalent. (Note that the use of identity tags is an effective mechanisms for dealing with the most common and serious difficulty resulting from the lack of logical links between the possible sets of bit patterns associated with program Values; section 3.2.2 discusses more general aspects of this problem and mentions possible means of addressing these difficulties.)

4.2.2.2 Size Information

As noted above, a Value represents some particular bit pattern or set of bit patterns that exist at run time. All of these bit patterns are of some fixed size. Moreover, in the current version of PARTICLE, such Values are not only of *fixed* but also of *known* size – that is, the system never manipulates Values whose size is not precisely known.[3] Moreover, there is currently no need for the size of a Value to ever be anything except an integral number of bytes (all bit field manipulations in the original C source code are desugared into integer operations by the point at which they reach the program specializer), but the system could relatively easily be extended to situations where Values are of arbitrary size.

4.2.2.3 Bytewise Decomposition Information

One of the requirements of performing program specialization on a truly low-level imperative language is the need to correctly and effectively handle overlapping writes and reads of different sizes. That is, if we write a value of a certain size, such languages may allow subsequent reads that may read all of or only part of the value, possibly in addition to some other neighboring information. In addition, programs can legally perform writes that may only partially overlap previously written values. To conservatively model such operations while allowing for the preservation of as much information about the associated Values as possible, every Value is internally broken down into "canonical" bytewise subcomponents. When we simulate the writing

[3]It is possible that the mechanisms needed to handle compound data structures of unknown size (such as arrays dimensioned using values unknown at compile time) would require the handling of values of unknown size. PARTICLE does not currently support such structures, although they represent one of the most significant lacunae in the current system.

```
...

y = x;

...

/*  Even if x not precisely known, value identity information allows us
    to recognize that the following predicate is false */

if (x > y)
        {

        ...


        }
```

Figure 4-1: In this case, value identity records the fact that x and y are equal despite the fact that their exact values are unknown. Knowledge of the equality between these two program quantities allows recognition of the fact that the predicate above is *false*.

of a value to memory, we actually write each such byte in turn to the appropriate locations, and record the association of these bytes with the complete Value we have just written. When we go to read a Value, we construct a string of these "Byte Clipped Values" by reading the byte in each location in turn, and check for the association of these bytes with any previous Values (in other words, check to see if these Byte Clipped Values represent the bytewise decomposition of any existing Value. In the usual case where we are reading a single previously written Value, we find the appropriate Value association and use that as the result of the read. In the case where we are reading some string of bytes that was not previously written as a unit (e.g., if we are reading a *subpart* of a previously written Value, or a string of locations that spans two previous writes), we must create a new Value, with the "Byte Clipped Values" as its canonical bytewise decomposition. We can use the knowledge associated with each of the Byte Clipped Values to dictate our knowledge of the new Value as a whole. For example, if the Byte Clipped Values are definitively associated with known values, a new exactly known value can be created. In cases where the Byte Clipped Values are less well known, it may be possible to construct a set- or range-based Value approximation. Although it would require only moderate effort to support such a system, the incremental benefit associated with this capacity for program analysis in practical programs lies far below the incremental benefit offered by other perturbations to the current system, and PARTICLE does not currently contain the mechanisms to deduce a tightly bounded Value approximation from a sequence of Byte Clipped Values.

The Byte Clipped Values stored with each Value represents a canonical decomposition decomposition of the Value into byte-sized chunks that exactly capture the bytewise information about program quantities and preserve enough information to permit the recovery of high quality information about Values read from memory even in cases where memory reads and writes take place at different granularities and on different boundaries.

4.2.2.4 Value Attribute Information

This information is designed to allow PARTICLE to maintain information about characteristics of Values that are not deducible from other information maintained within the Value abstraction. In the current PARTICLE experiment, this information is just used to track the propagation uninitialized (or *possibly* uninitialized) values. The maintainance of such information allows warnings to be issued to the user at points where the program may be relying upon an uninitialized value.[4]

4.2.2.5 Representation of Knowledge About a Value

The notion of a generalized value is conceptually distinct from *what we know about the run time value or values (bit patterns) associated with that generalized value.* Our knowledge of a particular Value can change over the course of time, but the Value itself is unchanging. We could imagine two Values about which we know the same amount of information (perhaps nothing!), but whose associated run time values could be quite different. Conversely, there could be two run time values that are associated with precisely the same bit pattern at run time, but about which the symbolic evaluator knows very different amounts of information. To observe this distinction, within PARTICLE the *representation of a Value* is distinguished from *representation of knowledge about the run time values associated with that Value.* (For example, the representation of a Value includes the identity information discussed in section 4.2.2.1 – this represents knowledge about a Value, but not knowledge about the possible run time values associated with that Value.)

The representation of the knowledge about the possible run time values associated with a Value makes use of the abstract domains whose underlying philosophy was discussed in section 3.3. PARTICLE makes use of increasingly less precise abstract subdomains of four different types: Subdomains to represent exact knowledge about a Value, knowledge that a Value is always a member of some discrete set of bit patterns, knowledge that a Value is a member of some numeric range of possible values, and a subdomain to represent the total lack of knowledge about a Value's run time character. Each of these subdomains is quite simple but deserves brief individual mention.

[4]It is possible to envision a system in which the input program is deemed to be so well-behaved that the use or reference to an uninitialized value in a path of execution would indicate that the path is *not* a valid run time path. (Similar reasoning is currently optionally used to terminate as impossible paths that involve moving pointer pointers outside the range of their initial segment – see section 4.2.3.2.)

4.2.2.5.1 Exact Knowledge The subdomain expressing exact knowledge of the bit pattern associated with a Value allows us to capture the information that a given Value is known to be some precise, concrete value; thus, it entirely captures the domain of all values manipulated in a standard language interpreter. Since the optimizations associated with program specialization often require exact knowledge of some Value (whether it is the knowledge of the result of a subexpression, the result of a predicate in a conditional statement or loop termination condition), and because more sophisticated domains such as sets and ranges make internal use of values drawn from this domain (to represent set elements or range endpoints), this subdomain plays an extremely important role during symbolic evaluation and analysis. Note, however, that not all exactly known Values correspond to (directly) "expressible" values – it is not always the case that a Value about which we have exact knowledge can be *trivially* translated into an expression. For instance, code that manipulates structures may frequently be working with structures whose contents is completely known. Unfortunately, in C there is no way of directly expressing a structure of known contents as an expression (e.g., in the way one might express the signed integer one by the expression "1"). As a result, there is no straightforward way to perform program specializations such as fetch elimination (constant propagation) on expressions involving structures.

Exactly known scalar Values are associated with a sequence of bytes that describe the precise values associated with each byte of the Value. (e.g., a Value associated with the floating point constant ".001" would be encoded in 4 bytes in accordance with IEEE floating point encoding conventions). Maintaining this information in a central place (rather than distributed among the Byte Clipped Values) allows for quick manipulation during operations that make use of this information (e.g., adding two exactly known values).

Although they are first-class citizens of the value domain in all respects, the representation of exactly known pointers is different than that used for exactly known scalar values. In particular, in most computations there is no need, desire, or even ability to represent the particular numerical value associated with a run time pointer value. Conceptually, the referent of a pointer can be *exactly known without knowing the particular numerical value associated with that pointer.* – all we need is some manner of unambiguously denoting exactly which to which lvalue the pointer refers. In addition to this information, all exactly known pointers are also associated with high-level segment information that is not maintained for scalar values (see section 4.2.3.2.).

4.2.2.5.2 Knowledge that a Value is a Member of Some Set While the symbolic evaluator cannot always discover a precise run time value for a particular Value, frequently it is capable of specifying the bit patterns possibly associated with that Value up to membership in some set of other possible Values[5] It should be stressed that set domain is not constrained to representing a Value as possible sets of *exactly known* values (sets of possible bit patterns) – the set of possible Values may contain Values that are fully general in character.[6] Figure 4-5

[5] Note that because of the strict ordering of all Value creation, recursive Value definitions cannot arise.

[6] It seems rather possible that the full generality of this representation is not needed and that virtually all of the benefits of the set representation would be offered by a representation that restricted set membership to exactly known values. Adopting such a policy could cut down on needless waste of space arising from the representation of sets of arbitrary Values and might make the maintainance of a more generous maximum set size practical: See 3.3.4.

illustrates a common case in which a Value can be determined to be a member of some set: Consider a program fragment containing a conditional, where one branch of the conditional contains the assignment of one Value to a variable, while the other branch assigns another Value to the same quantity. If there is insufficient information to evaluate the truth value of the predicate of the conditional, the symbolic evaluator will be unable to determine which branch of the conditional will be taken at run time. Because either branch could be taken and each assigns a different Value to the variable, the symbolic evaluator will not know the precise Value of the variable after the conditional even if the Values assigned in each branch are themselves fully known. However, it will be able to determine that the resulting variable value is one of the two Values assigned to the variable inside the conditional. This knowledge can be expressed by specifying that the value associated with the variable is a member of the appropriate two-set of Values.

Sets of values can be produced during symbolic evaluation by four major mechanisms:

- By the generalization of a set of states at a program join point (including the repeated generalization performed during loop and recursive function abstraction. (See section 4.2.3.1.).
- By the specification by the user that some input Values are to be taken as a member of a certain set of values.
- Through the mechanism of *weak writes*: In cases where the exact lvalue of the location being written to by a pointer write (or a write to an array element) is not known, the symbolic evaluator must create a model of the resulting state that represents a legal approximation to *any* of the states that could possibly result from the write. The symbolic evaluator accomplishes this by updating the contents of any lvalue that could be affected by the write. The lvalue is updated in such a way as to express the fact that it *may or may not* have been changed. If a particular lvalue that may have been overwritten previously contained a Value v_1 and is *possibly* overwritten by a value v_2, a conservative approximation to the Value associated with the lvalue following the write would be a Value associated with the set of alternatives $\{ v_1, v_2 \}$ (reflecting the fact that the lvalue may have kept the same value, or may have been overwritten).
- Inductively by operations on some preexisting set Value (e.g. negating a Value associated with a set approximation will yield a new value with a set approximation. Similarly, the addition of a Value approximated by a set and a fully known Value will yield a new set Value).

As discussed in section 3.3, PARTICLE allows the user fine control over the richness of the abstract domains used in the process of symbolic evaluation, thus providing the user with a means of varying the height of the domain and providing a means of capturing graceful degradation in knowledge about a Value. This control is exercised by increasing or decreasing the maximal allowable set size used during program analysis. In recognition of the fact that it is frequently desirable to have a very high-fidelity representation for pointers even when other values are simulated less precisely, PARTICLE allows the user to separately specify the maximal set sizes associated with sets for pure pointers and those associated with scalar values. Moreover, PARTICLE allows the user to separately specify distinct maximal set sizes for sets produced in

```
x = a;

if (x > y)              /* unknown predicate */
        {
        x = b;
        }

/*  x is known to be EITHER   a  or b */
```

Figure 4-2: Origins of Set Approximations: Program Join Points. Lack of knowledge about the truth conditional of the predicate in the code fragment above renders the system unable to determine whether the conditional will be taken at run time. While the system will not know the exact value associated with x after the conditional, it *will* know that x is associated with the value of *either* a or b.

```
Enter the Number of the Jth Bessel Function to Evaluate:   { 4, 5}
```

Figure 4-3: In order to permit fine-grained specification of external program input, the user is permitted to specify sets of possible values in response to requests for such input.

```
/*  p imprecisely known */

*p = x;
```

Figure 4-4: When PARTICLE encounters a write to memory whose lvalue is not exactly known, the contents of lvalues that have possibly been written are associated with a set of possible values that includes their previous value and the written value.

```
/*  p imprecisely known */

x = *p;
```

Figure 4-5: Sets of values are frequently created inductively by operations upon preexisting sets of values

three different ways: Through computation from pre-existing sets, and through the unioning or abstraction of several existing Values. The ability to separately specify lattice heights limits for each of these spheres of activities permits the user to adjust the tradeoff between compilation time and quality of symbolic evaluation in a manner that allows the maintainance high fidelity in certain spheres of activity, but permits less high quality representations to be used where compile time performance is important (e.g., during abstraction of loops and recursion). Finally, PARTICLE provides the option of "flattening" all knowledge representations consisting of nested sets-of-sets; this option permits somewhat faster processing of such Values, but has the effect of eliminating the identity tags (see section 4.2.2.1.) information for the intermediate Values which are flattened. At this point in the project, it is unclear whether the full range of controls provided are truly necessary and useful; it may well be the case that a system without the full range of "knobs" listed above would be sufficient for the specialization of virtually all programs. It is hoped that further experimentation with the use of this mechanism in realistic program specialization will help to answer this question.

4.2.2.5.3 Membership in Ranges

4.2.2.5.3.1 Overview Although the set representation offers a great deal of precision in specifying the possible run time values that could be associated with a given Value, the size of the representation grows linearly with the number of alternate bit patterns for the Value, and the cost of performing binary operations on such values can grow with the square of the number of alternatives for each operand (see section 3.3.3.). As a result, users of the program specializer concerned about compile time performance may be forced to seriously restrict the maximum set size. When the size of a newly created set representation exceeds the user-specified maximum set size for the operation that created the set, it would be ideal if the symbolic evaluator could have a means of gracefully converting to a less expensive representation, but one that would still capture a great deal of information about a Value. As an intermediate point in specificity of knowledge between a set representation of possible Values and no knowledge at all, PARTICLE makes use of a subdomain that identifies a Value as a member of some numeric range of values. The range can be associated with any arithmetic type of value that has same size as the quantity of interest. (Note that such a representation includes as special cases representations of knowledge such as "a positive integer", "a floating point value greater than 100.0." Each of these representations can be achieved through appropriate choice of the range endpoints.) For example, on a certain machine, we might identify a program quantity as being a member of some range of floating point numbers (e.g., in the range $0 - 1.0$). Subsequent arithmetic operations on this Value will not yield results as precise as if the Value were approximated by a set expressing the precise alternatives for the Value of the quantity, but the results of operations on the range will be much easier to compute (since to compute the outcome of many operations we need only compute with the endpoints of the range, whereas every member of a set must be separately operated upon), and much more compactly expressed.

In light of the fact that a Value is merely a representation of an untyped set of possible bit patterns, it may seem entirely off-base to make use of a numeric range to express the system's knowledge of the possible alternative bit patterns. After all, the very idea of a numeric range has non-trivial semantic underpinnings in that it requires that some complete ordering be imposed on upon a set of bit patterns. While this policy seems to be in violation of the philosophy of maintaining the notion of Value free of any "type", in fact it is not. The key point is that while the Values are inherently untyped objects (merely representing possible patterns of run time bits), the *representations used to encode the knowledge about the possible bit patterns (values) associated with a Value need not themselves be untyped.* In the case of ranges, for the sake of compactness we have chosen to approximate a set of possible values using some representation that represents a *superset* of these values. The means used to *express* the enclosing superset is irrelevant to the character of the values being approximated. All that is guaranteed is that the values being approximated are members of the specified superset. When seen in this light, a range is simply a compact and convenient manner of expressing that a Value is a member of some set of bit patterns.

4.2.2.5.3.2 Selection of Range Type The type chosen to represent a given set of values is can be chosen arbitrarily, as long as

- The run time representation of a value of type of the range is of the same size as the Value to be represented.
- The type of the range is associated with some total ordering (e.g., a record type for the range would not be reasonable – there is no notion of one record being "greater" or "lesser" than another).
- We pick two endpoints for the range such that all of the values in the set are greater than or equal to the lower bound of the range and less than or equal to the upper bound of the range *when the members of the set are interpreted as quantities in the type of the range.*

It is important to stress that the concept of a range is meaningful only in the context of a total ordering imposed by association of the range with a given type. The members of the range are simply *bit patterns interpreted in the context of the associated type.* Since the interpretation of the magnitude of a given bit pattern may be wildly different for different types, a range from the point of view of one type may not even correspond to a contiguous block of numbers from the point of view of another arithmetic type, much less to a monotonically increasing set of numeric values.

Given the above constraints on the possible types used in constructing a range representation for a given Value, there are certain criteria of desirability that we would like to fulfill. Such criteria will be used to guide us in the selection of a type for the range among the set of all possible such types.

- We wish to make use of a range that is as "small" as possible from the point of view of the values of interest to the program. In a sense, we want the "size of the range as it appears to the rest program behavior" to be as small as possible – to make the range *look* like as close an approximation to the actual set of values as possible. The explanation of this criterion is provided below and is slightly involved; readers not interested in the details of the justification may wish to bypass this section.

  The idea behind the criterion of choosing a type to minimize the "dynamic cross section" of the range being created is that the program may perform various relational operations on the value represented by the range. These relations could include tests to see if some other given value is equal to the value represented by the range, or comparisons *in some specified type system* between the magnitude of the Value represented by the range and some other Value (which may be different than that for the range). Tests of equality for a range can never be answered definitively in the affirmative (since except for completely degenerate ranges we never know *which particular value* in the range the current Value corresponds to). At best, we can show that the Value being tested for equality with this one is not even *possibly* equal to this Value, a result that can be reached by demonstrating that the Value being compared is outside of the range associated with this Value. Thus, in the case of equality comparisons, we would like the range to be as "sparse" as possible with respect to values of interest to the program: We would like to have a range that would be the least likely to include any given point checked for comparison, but would still contain the set of possible bit values being approximated.[7] For relational comparisons such as "greater than" and "less than", however, it would be desirable to have a range that is as simple and compact as possible from the point of view of the *type of the relational comparison*. Were we to make use of a range which was (by definition) contiguous from the point of view of the range type but enormously sparse from the point of view of the type being used in the relation (as seemed desirable for equality comparisons), a relational operator such as "greater than" would be highly unlikely to be able to make any firm statements – just because from the point of view of the type associated with the magnitude comparison, the range would include values *both less than and greater than* the Value being compared to (where notions such as "less than" and "greater than" are interpreted in the context of the type under which the comparison is being made.) For the sake of such magnitude comparisons, we would like to make use of a range type that will lead to the range values corresponding to a simple object such as a range in the type in which the comparisons are being made.

- We wish the combination of the range with other Values to be as efficient as possible. Ideally, we would like the combination of the range with other Values to yield a result that is easily expressible as well (so that we can represent our knowledge about it as a range or using an event more precise subdomain).

In light of these two criteria for choosing an appropriate range type, the natural choice of type to associate with the range is *the type of the program quantity associated with the Value approximated by the range.* Such a range representation will allow precise magnitude comparisons in the common and expected case that such magnitude comparisons are made using the same type as associated with the program quantity, and will allow efficient combination of the Value in the expected case of combination with other Values of the same type (e.g., adding a fully known Value of the range type to the range will just shift the endpoints of the range; inverting the range in the arithmetic type for the range will simply lead to an inversion and reversal of the range endpoints, combination of one range to another will merely lead to a new range, etc.) Thus, while the choice of the type used in the interpretation of the range used

[7]In such cases, it might seem desirable to make use of a range type that would include few values that are close in magnitude to the value being approximated (where the "close in magnitude" comparison is made in the context of the type in which most equality comparisons against the Value approximated by the range will take place), under the assumption that if we have a value with a certain value, we are more likely to test its equivalence to values fairly close in magnitude than to values at considerable distance.

to approximate the Value is conceptually independent of the Value itself, it is convenient to associate a type with the range that also happens to be the type program quantities associated with the (inherently untyped) Value.

4.2.2.5.4 Complete Ignorance The final form of representation of program knowledge of run time values is used to describe Values about whose possible bit patterns we have no knowledge whatsoever. Values associated with this level of knowledge (or ignorance) are treated as approximations to *any* possible bit pattern of the appropriate size. (Recall, however, that such Values can still have *particular identities* that allow the program specialization to establish equivalence between them.)

4.2.2.6 Values: Concluding Remarks

The discussion above has sketched the fundamental character of the Value abstraction that is used to represent run time values in PARTICLE. Values are one of the central quantities manipulated by traditional program interpreters, and their approximation is one of the most important abstractions used in PARTICLE. Through the use of a multi-tiered and arbitrarily extensible set of abstract value domains, PARTICLE permits high quality simulation of program quantities, while allowing the user fine precision and granularity of control over tradeoffs between quality of program specialization and the run time of program analysis. We turn now to examine another central abstraction in the PARTICLE framework: The model of memory.

4.2.3 Memory Model

4.2.3.1 General Structure

At every point in the run time execution of a program, the program is associated with some particular state. The primary job of traditional program interpreters for imperative languages is the simulation of each statement of the program as it maps input states to output states. As has been discussed in previous sections, symbolic evaluators for such languages are responsible for performing the generalized version of this process: the simulation of each construct (statement, declaration, etc.) as it maps from one approximation to program memory state to another during the execution of the program. Unfortunately, the effective performance of this job within symbolic evaluators is considerably complicated by the fact that within such systems the exact pattern of *control flow* within a program may not always be fully known (e.g., it may not be known which branch of an *if* will be taken at run time, or how many iterations of a loop will be executed.). As discussed in chapter 3 and in section 4.3.3.2.1.2, at times the lack of knowledge about exact control flow from a point forces the evaluator to model the results of *all* possible paths of control flow. This is accomplished by the *generalization* of the results of taking each of these possible paths together to form a state that is guaranteed to represent conservative approximation to the actual run time state *no matter which path or set of paths will actually be taken at run time.* In order to collect together the possible results of each separate path (in order to generalize them together), we are forced to execute each such path in turn. Because each such path must start out from the program state that obtains at the "start" program point and because we must have some means of storing the generalized results of previous paths during, performing generalization requires managing the simultaneous coexistence of many different models of memory during symbolic evaluation.[8] As a result, the symbolic evaluator cannot simply maintain a "global program state" (as would be done in a program interpreter) – we must be able to simultaneously (if temporarily) maintain several independent states simultaneously, in order to represent the states resulting from different paths from the "forks" that arise when encountering a point after which control the pattern of control flow is uncertain. (See figure 4-6.)

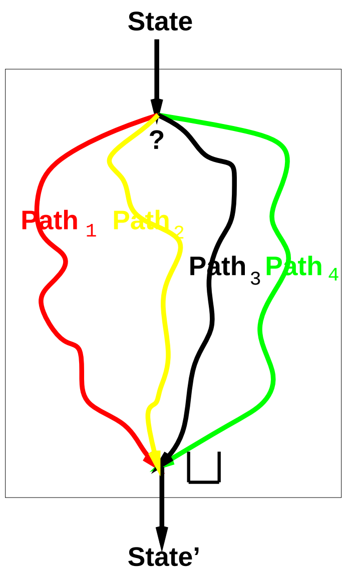

Figure 4-6: In order to conservatively simulate the behavior of the program in instances where run time control flow is uncertain, PARTICLE must simulate each distinct path of control flow and create a legal approximation to *all* possible resulting states.

While a naive solution to this problem would involve the maintainance of an evolving global state that is copied whenever a "snapshot" is needed (e.g., as a state to which to roll back or to represent the result of a path of execution), this approach would carry with it an unacceptably high performance overhead: In many pieces of code, we can expect to have a large fraction of the conditionals dependent on predicates that evaluate to unknown Values, and the need to abstract loops is equally common. Each of these operations requires copying the entire memory state for each such event could require a substantial and highly expensive series memory allocations. Moreover, independent branches of execution need to be joined at subsequent program "join" points; copying the entire model of memory would require either the exhaustive merging of *all* portions of that memory model or the maintainance of a bookkeeping structure that would help to locate the changes in content between two memory models.

A substantially more desirable solution can be arrived at by recognizing the fact that typi-

[8]The need to simultaneously maintain, manipulate, and combine several program states also arises in the process of having to do with the conceptually similar process of "abstraction" (see section 4.3.3.2.1.2.) needed to approximate within a finite time the effects of performing arbitrarily many iterations of a loop.

cally there are rather few changes between one simulated program state and the next, and that frequently the changes observed over the course of the execution of a conditional or loop affect a rather small portion of the entire system state. Rather than simultaneously maintaining several disjoint snapshots of different points in the evolution of system state, we can instead maintain a single, global memory model that contains within it information about *all* simultaneously existing memory models. A given write to the memory model annotates the locations affected with the new Value and a tag indicating the "name" of the state at which point the change is effective. Evolution of the memory model thus proceeds by the means of successive recording of small changes from one state to another. If we wish to read a given location *in the context of a given state*, we need to find the particular annotation that dictates the Value of that location for the desired state. From the point of view of the state at which we desire to do the read, this is the annotation corresponding to the most recent write to that location.[9] We can find the annotation associated with that write by stepping back on the path of previous states leading up to the state from which we wish to read the value, and finding the first such state that is associated with an annotation. (See figure 4-7.) Such a design for the memory model allows a means of simultaneously capturing many different program states in the same structure, and exploiting the sparse character of change between program states. Through the use of simple bookkeeping, we can eliminate the cost of maintaining unneeded annotations by garbage collecting annotations corresponding to writes that are entirely eclipsed by later changes (such as writes or other modifications arising before program join points).

Finally, recall that the generalization of two memory models is performed simply by the location-wise generalization of the contents of those memory models. Since the generalization of any two Values about which the same information is known merely yields the same Value, the generalization of two arbitrary memory models yields a memory model that is precisely the same as both of the memory models where they are equivalent and differs from these memory models only in the contents of the locations where those two approximated states themselves differ. This observation permits a rapid means of generalizing two memory models: The generalization can be found by basing the generalized memory model on one of the memory models to be generalized and generalizing just those locations whose contents have changed. The "change-oriented" structure of the memory model presented in this section allows easy implementing of mechanisms to rapidly find the an upper bound on maximal set of possible locations that have changed between two memory models. (We can find such a list of possible changes by unioning together the set of all locations that have changed between the common ancestor to two program states and the states themselves).

4.2.3.2 Memory Model Segments

The segmentation of a machine's memory permits the establishment of local, independent name spaces, and provides a means of protecting machine resources from the deliberate or accidentally destructive behavior of user programs, and rules out certain categories of program behavior as illegal (e.g., performing writes into one's code segment). PARTICLE makes use of a similar mechanism to simulate run time segmentation of different granularities; because segmentation

[9]In order to handle the boundary case by ensuring that we always have have a preceding write, we can treat the creation of that location as a write.

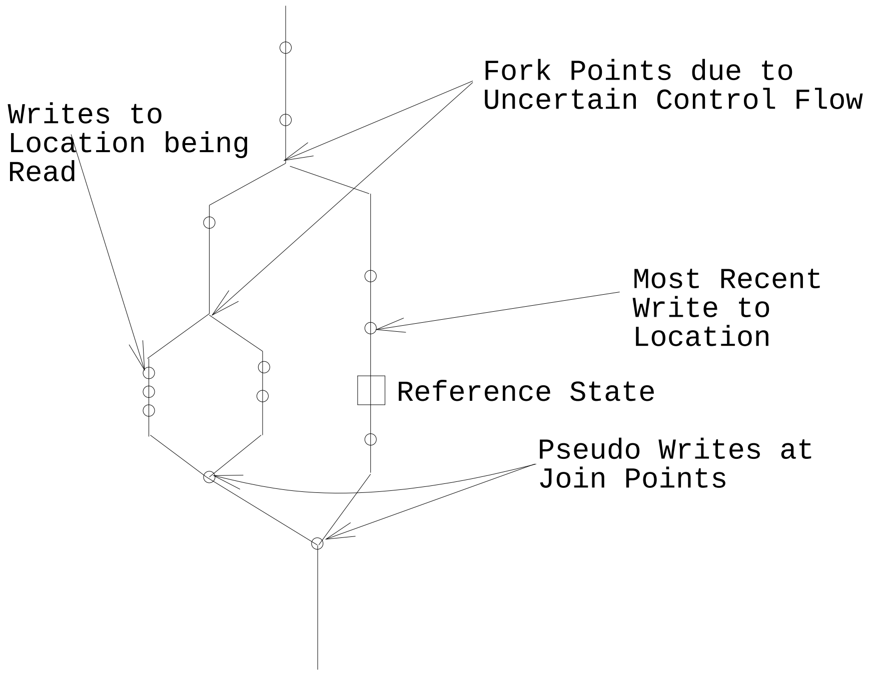
Writes to Location being Read
Fork Points due to Uncertain Control Flow
Most Recent Write to Location
Reference State
Pseudo Writes at Join Points

Figure 4-7: This figure illustrates PARTICLE's maintainance of simultaneous memory models corresponding to different execution points in the source program. PARTICLE perform a reads of a particular memory location in a specific memory model by finding the state associated with the most recent write to that memory location (where the notion of "recent" is taken *relative to the memory model being read*) and recovering the recorded value from that state.

imposes semantic constraints on legal program behavior, such mechanisms can permit improved program analysis.

4.2.3.2.1 Coarse-Grained Segments Running user processes are often associated with the different data segments associated with heap, stack, and global data; PARTICLE's model of machine state is divided in the same manner. Each of these segments is regarded as entirely independent of the others, and all precisely known program pointers (both user-created and implicit pointers such as the stack pointer) are associated with a single such segment.[10] Although it might prove useful for limiting the disastrous effects of writes to pointers whose referents are unknown, PARTICLE currently does not support any manner of encoding the knowledge that the referent of a Value is entirely unknown, but is known to reside within the span of a given coarse-grained segment (in almost all cases, *far* tighter bounds can be established for the possible referents of a pointer, bounds that record the fact that the pointer refers to an element

[10]In PARTICLE, Values approximated by sets of possible pointer Values and ranges of possible pointer values are indistinguishable from ranges or sets of other values (except for by appeal to the character of their endpoints or set membership), and have no particular segment affiliation, although the endpoints of ranges of possible pointer values must belong to the same segment.

of some array or points to some set of possible locals). Within PARTICLE, coarse-grained segments serve to partition the name space of data locations into its logical domains, and do not currently offer any benefits for program analysis. We turn now to a discussion of fine grained segmentation within PARTICLE, whose fundamental purpose is to allow serve as an aid in static analysis by allowing tighter bounds to be placed on the possible referents associated with a pointer.

4.2.3.2.2 Fine-Grained Segments and "Sanity" Assumptions In addition to supporting segmentation of the program memory model along the very coarse-grained lines discussed above, PARTICLE also provides the user the option of assuming segmentation at the granularity of array spans. Such assumptions permit the program specializer to assume that if one begins with a pointer into the array and adds (or subtracts) some unknown offset, the result represents a legal pointer to some element of that array (as opposed to being conservatively modeled as possibly now pointing the array or pointing to other program quantities that might precede or follow the array in memory). The example above is a general illustration of a class of assumptions of "program sanity" that PARTICLE uses by default but allows the user the option of overriding. It should be noted that existing optimizing compilers may make extensive use of similar assumptions when performing aggressive program analysis [2]. In many cases, the application of such assumptions can make an enormous difference in the quality of the program analysis and optimizations performed. (For example, without such assumptions a simple array write with an unknown index must be considered as possibly overwriting any possible value in the stack and thus possibly modifying all existing local variables, parameters, and continuations for function return). To permit occasional violations of such assumptions, it would be highly desirable to have a mechanism for fine-grained specification of when "sanity assumptions" are permitted when they must be suppressed. Through the general mechanism of external calls from the code to the PARTICLE system (see section 4.3.5.1.2.), PARTICLE supports spatially precise enabling and disabling each of the different "sanity" assumptions.

Note that that "insane" code frequently relies on certain implementation details not specified by the language (e.g. assumptions concerning the arrangement of local variables and arrays on the stack); while PARTICLE is not obliged to support such assumptions, doing so requires little work. In other cases, the application of the term "insane" to code that breaks the above rules can hardly be justified. For example, frequently pointers are employed as cursors to scan through an array. In the process of performing several several subsequent scans through the array, the pointer may transiently leave the bounds of the array (e.g., at a boundary case just before reversing direction), although it may never be *written or read* when outside of the array. Unfortunately, because of the transient departures from the array this code violates the assumptions made concerning legal program behavior. Such uses uses of a pointer can hardly be characterized as "insane" and the assumptions employed by the system about program behavior should be correspondingly loosened.

4.2.3.3 Locations

The fundamental unit of storage in the memory model for the PARTICLE system is the location. Conceptually, a location is associated with a single Byte Clipped Value for each system state in which it is exists, recording the contents of that location in that state. As discussed in section

4.2.3.1, the storage requirements demanded by the PARTICLE system are far more sparse: Internally, the location contains data only for each write to this location, and the illusion of complete internal representation of values for all system states is maintained by finding the most recent write for any specified system state.

When the program reads a contiguous set of locations, it tries to piece together the bytes into a recognized preexisting Value; if this attempt fails, it creates a new Value to represent the quantity read.

4.2.4 Data Structures for Specialization

The abstractions described above are all central to the symbolic evaluation of the user program. Thus, they are used in the collection of information on possible specializations for the input program. Another class of abstractions embodies this information about potential specializations, and play central roles during code generation.

4.2.4.1 Annotations

4.2.4.1.1 Construct Annotations Every expression and statement in PARTICLE that is subject to being specialized is associated with a set of *annotations*, each of which is used to indicate a specialization that is currently perceived as legal for that construct in the context of a given call to the enclosing function. This annotation specifies the particular specialization with which it is associated, the call with which the specialization is associated, and an estimate for the number of times the current construct is executed during that call (in order to help estimate the savings associated with the specialization). The specialization associated with the annotation is specified by (and, at code generation time, is performed by) the *reduction* abstraction (see below).

4.2.4.1.2 Function Annotations In every function in the code to be specialized, PARTICLE associates information indexed by different calls to that function. For a given call, this information describes

- The estimated number of run time invocations associated with that call. While for many calls this number will be one, if the call takes place in the process of "speculatively" evaluating some construct whose run time control flow is not known (e.g., a loop whose body will execute an unknown number of times or a conditional whose predicate evaluates to an unknown truth value) the estimate may be less certain but smaller or larger than one.

- A record of the statements that have been evaluated for the current function and call. When formulating the specializations that are legal for a given call site or group of call sites, this information allows us to perform careful intersections of legal optimizations. In particular, a specialization is normally legal for a given call site iff it has been recorded as legal for each of the recorded calls from that call site. At a higher level of abstraction, a specialization is judged as legal for a given specialized function iff it is legal for all of the call sites that will make use of that specialized function. But it may be that a specialization is legal for a particular call or call site without being explicitly recorded as such. In particular, if a given call or call site never executed the statement associated with the specialization, then while we will have no annotations concerning possible specializations for constructs associated with this statement, we know that since our symbolic evaluation implements a conservative approximation to the patterns of control flow seen at run time, the call or call site under consideration is guaranteed to never execute this statement at run time. As a result, we can safely do *anything* to that statement without affecting the behavior of the function for that call or call site – certainly, we can legally apply a specialization to it. Thus, a specialization is legal for a given call either if the associated statement was executed for that call and the specialization was noted as legal during all evaluations of the statement that call *or* if the statement was never executed during that call. Generalizing this observation to a higher level, a specialization for a construct associated with a given statement is regarded as legal for a call site iff it is legal for all calls from that call site which *actually executed that statement.* Similarly, a specialization is regarded as legal for a group of call sites iff it is legal for all call sites in that group for which the statement was actually executed. The record of statements that have been executed for a given call allows us to check this condition.

- A record of the reductions explicitly judged legal for this call. Note that for performance reasons this list is collected from the program constructs after symbolic evaluation has terminated, and is kept in a centralized manner. As described above, this information is then used to dictate the sets of reductions that are legal for a given call site.

4.2.4.2 Reductions and Reduction Templates

4.2.4.2.1 Reduction Templates The central abstraction used in the representation and execution of program specializations is the *reduction template.* A particular class of a reduction template embodies a particular kind of specialization for some class of construct. A particular *instantiation* of a reduction template represents a particular instance of this specialization (e.g., "replace the binary expression by the expression '3'") There are a range of different types of reduction templates: different types of templates for classes of statements, and for different coherent groups of expressions (e.g. binary vs. unary operator), but all reduction templates share certain commonalities. Reduction templates are created whenever the opportunity for a specialization is first discovered during program processing. Although the creation of a reduction template is extremely simple and can be performed at a number of different levels of specializer function, typically this occurs in the machinery to simulating operations on Values (for expressions) or in the process of simulating the effects of a statement (for statements). At these points, PARTICLE will discover that the situation permits some particular specialization to be applied. For instance, during the evaluation of an expression, we may find that the expression has evaluated to a constant value, or that the left operand of a multiply always evaluates to 1 so that the result is equivalent to the (unknown) right value (a discovery presumably made at a somewhat different point). In a statement, we may discover that the predicate of an *if* statement is *true*, so that the predicate check and the alternative can be eliminated, or that a while loop should be unrolled. A complete taxonomy of the rather compact set of specializations supported by PARTICLE can be found in figure 4-8. Each of these specializations has an associated reduction template, parameterized in the appropriate manner to allow use in all cases where the specialization will be needed (e.g., the replace-binary-expression-by-constant-expression reduction template is parameterized by the constant expression with which to replace the original expression). During code generation, the reduction template accepts as input specialized subconstructs (expressions or statements) and the original construct and outputs a specialized version of the original construct as applied to these specialized subconstructs. For instance, a reduction template associated with the replace-binary-expression-by-constant specialization would accept both of its proposed subexpressions and a copy of the original construct and discard them, outputting only the appropriate constant expression. A collapse-if-consequent specialization would accept its specialized predicate and specialized substatements for the consequent and alternative, returning the alternative.

- Program level specialization (specialization of a program with respect to external input).
- Function level specialization (specialization of a function with respect to different calling contexts).
- Statement level specialization
  - While Statement
    - Collapse statement
    - Unroll statement
  - Conditional Statement
    - Collapse conditional
    - Collapse alternative
  - Collapse Expression Statement
- Expression level specialization
  - Unary expression
    - Collapse to constant
    - Eliminate operator and pass through specialized operand
  - Binary expression
    - Collapse to constant
    - Eliminate operator and pass through specialized left operand
    - Eliminate operator and pass through specialized right operand
  - Collapse function call to constant
  - Collapse leaf expression (typically variable reference) to constant

Figure 4-8: A taxonomy of the small set of specializations supported by PARTICLE.

In this phase, the reductions chosen for application within each specialized function (and their internal reduction templates) accept as input already specialized subconstructs (expressions or statements) to their construct, and output a specialized version of the original construct as applied to these specialized subconstructs (for instance, a reduction template associated with the replace-binary-expression-by-constant specialization would accept both of its proposed subexpressions and discard them, outputting only the appropriate constant expression).

In addition to representing and performing specializations, reduction templates are associated with sufficient information to allow to estimate the savings associated with the reduction they perform given cost information concerning their subexpressions or substatements. This information can be used during function specialization decisions to help weigh the benefits and costs associated with making use of one set of optimizations rather than another. (See section 4.4.).

Thus, particularly instantiations of reduction templates represent particular specializations and are capable both of performing those specializations and of estimating the costs and benefits associated with the specializations for a given execution of the surrounding function.

4.2.4.2.2 Reduction At some point following its creation, a reduction template (which by itself simply represents some particular specialization) is incorporated into a *Reduction* – an abstraction which associates the reduction template with some particular construct (always an expression or statement within the current PARTICLE system). As described in section 4.2.4.1, this reduction is then used to annotate the construct associated with the proposed specialization to express the existence of such a specialization for some particular call.

After program analysis terminates, the sets of legal reductions for each call sites are computed, and function specialization decisions are made on the basis of the overall costs and benefits of creating new functions; as part of this process, reduction templates are used to estimate the time and space savings associated with different specializations. Finally, as discussed above, in the code generation phase the reductions and reduction templates take on their primary dynamic duty: Performing the reductions they represent.

4.2.5 Conclusion

This section provided a brief introduction to the most important abstractions used in PARTICLE. Discussion now turns to a high-level overview of the central algorithms associated with these abstractions.

4.3 Algorithm: Analysis Phase

Earlier sections have sketched the fundamental manner in which the system performs program analysis by means of symbolic evaluation. The aim of this section is to flesh out the algorithms and mechanisms used during this process.

4.3.1 System Operation

The overall operation of symbolic evaluation in PARTICLE is simple: The system steps through the program, with each statement or declaration mapping input state to output state and type declarations inserting appropriate names into the global namespace. Within a particular expression (always functional within PARTICLE), the input state is established as the context state and the expression evaluation takes place within that context. Any specializations discovered at the statement or expression level during this process are recorded for processing by the second phase of program operation (specialization).

The following sections discuss symbolic evaluation at the expression and then the statement level.

4.3.2 Expression Level Evaluation

Previous sections have outlined the basics of the symbolic evaluation of expressions, and this section reviews and expands upon those earlier descriptions.

4.3.2.1 General Functioning

Expressions within PARTICLE are guaranteed to be purely functional (all assignments and function-calls take place in isolation at the top level: See section 3.5.4.), and can thus be evaluated within a single execution context. Expression evaluation is performed recursively, in a manner that mirrors the evaluation of expressions in program interpreters (i.e., in order to evaluate itself, an expression evaluates its subexpressions and combines the results in the appropriate manner). (See figure **??**.) While expression evaluation in traditional interpreters yields concrete values, information returned from a given expression evaluation in PARTICLE consists of a bundle of four quantities: A reduction template indicating any specialization that were discovered for the given expression, a cost estimate for the evaluation of the subtree rooted at the expression (to permit specializations further up in the expression tree to get an estimate of how much savings they permit by e.g., eliminating the computation associated with this subtree), a type associated with the expression result[11], and the Value itself. Immediately following the evaluation of the expression rooted at a node, the reduction template is promoted to a reduction and associated with that node if it is compatible with any previous reduction specifications. The following section describes the workings of this process in more detail.

During evaluation, expressions call off to separate routines to perform the appropriate operations on the Values resulting from the evaluation of the expression operands. If desired by the user, these routines optionally check to see if the operation has been evaluated for identical operand Values in the past.[12] If a record of a previous evaluation of this construct is discovered, the information returned from the earlier evaluation is returned again. (See section 4.2.2.1 for a motivation for this mechanism.)

[11]Note that within PARTICLE, types are passed around during expression evaluation to avoid the need to allow for a form of polymorphism in evaluating operations. The current method is associated with unnecessary performance overhead, and should be replaced by a system in which the system maintains explicitly distinct operators for different typed versions of the same conceptual operation (e.g., so that there would exist a "float+", a "short int+", "double+", etc.)

[12]It should be stressed that such caching is feasible because Values are immutable and "timeless" objects; as a result, the return result of a given expression involving values will remain the same throughout the evaluation of the program. Note that the situation becomes more complicated once conditional contexts are introduced (see section 5.5.3.): If we can acquire and lose knowledge concerning a Value in the process of execution, the result of operating on that Value at one point in the program may *not* be the same as the result of the same operation at some other program point.

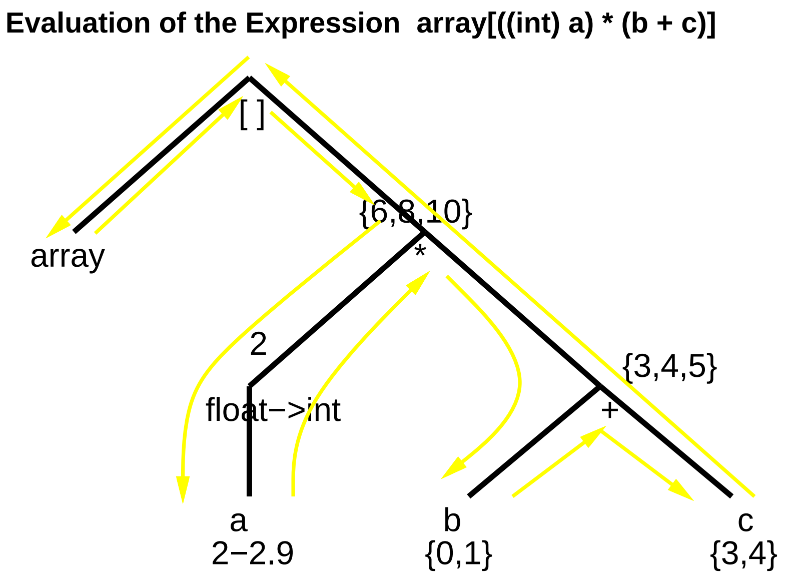

Figure 4-9: The symbolic evaluator's flow of control during expression evaluation mirrors that of a traditional interpreter: Just as operators within an interpretation combine together concrete values, operators within the symbolic evaluation of a program combine together approximations to values.

If the result of the operation to be performed is not already cached, the routines operating upon the values check if the operation to be performed can be accomplished just by making use of the Value-level information concerning the object identities of operands to the expression.[13] Such information can allow complete evaluation of expressions such as "$v_1 == v_2$" and "$v_1 - v_2$" without the need to inspect or manipulate the encodings of the degree of knowledge available about the possible sets of bit values represented by v_1 and v_2.

In the common case in which the identities of each of the operand Values alone are insufficient to allow evaluation of the operation, the symbolic evaluator creates a new Value by performing calculations on the information representing knowledge of possible value bit patterns. (See section 4.2.2.5.) These calculations take place upon elements of PARTICLE's abstract Value representation domains (see section 3.3.), and each operation must in general handle operands drawn from any combination of these domains. (e.g., A *plus* operation must be capable of handling left and right operands drawn from any combination of exactly known, set-based, range-based, and unknown representations of knowledge about a Value.)[14]

[13]Recall from section 4.2.2 that such information is maintained at the Value level rather than at the Value *knowledge representation* level (i.e., the level bounding the set of possible bit patterns associated with that Value): While from the representation of knowledge about the Values it may not be obvious that two Values are guaranteed to always represent the same run time value (e.g., if both are represented as "unknown" values), information about Value identity can establish such equivalence regardless of the degree of knowledge about the possible bit patterns for those values.

[14]As noted above, because of PARTICLE's current failure to explicitly create distinct operators associated

During the processing of the operation at either the Value or knowledge representation level it may be discovered that the some specialization is possible. For instance, it is possible that evaluation of the Values reveals that a binary operation reduces to a known Value and can thus be constant folded or that a binary integer multiply expression is associated with a operand known to be 1 and is thus reducible to the expression associated with the other operand. All such possibilities for program specializations (and their estimated time/space savings or costs) are encoded by appropriate reduction templates. In cases where no specialization is discovered to be possible, a reduction template communicating this fact is created. The reduction templates associated with this computation are returned along with the Value resulting from the operation and with cost information for the entire expression. This information is returned to the level of the expression that invoked the operation to be performed, where the expression is (potentially) annotated with the reduction information and the Value, cost and type information are recorded in the cache memoizing the results of expression evaluations on Values (if this mechanism is enabled) and are returned as the result of the expression evaluation (and presumably then are used in the context of some enclosing expression or statement, just as the Value, cost and type of the operands to the current expression were used within the evaluation of that expression).

Before concluding the discussion of the process of expression evaluation, we turn to briefly examine the character of the expression annotations created during this process to represent possible reductions. Because the mechanisms and philosophy of expression annotations are shared by statement annotations, the following discussion encompasses both types of abstractions.

4.3.2.2 The Creation of Expression and Statement Annotations

An attempt to record a construct annotations is made whenever an opportunity for optimization is discovered during evaluation of the construct. In order for a specialization of a construct to be legal for a given function call the specialization must be legal for *all* executions of the construct during that function call. Before recording a proposed reduction for a construct, PARTICLE checks to see if some different specialization (possibly just "no reduction") was already recorded for that construct during the current function call. If the system discovers that some different reduction has already been recorded, there must already have been another execution of the current construct in the context of the current call that did not allow the specialization (reduction) currently being proposed[15]. In such a case, we are forced to conclude that the construct can support no specializations (because there are is no specialization recorded as applicable for *all* executions of the construct). In all other cases (where we have not previously evaluated this construct in the current call, or where we have evaluated this construct at another point but discovered that the same optimization was applicable), we make an annotation in the construct noting the applicability of the current reduction. If this annotation is not canceled at a later evaluation of this construct during the current call, and if it is permitted by all other calls from the call site from which this call originates, the specialization associated with the annotation will be considered during the specialization and code generation phase as a potential optimization to be exploited by a specialized function. (See section 4.4.4.)

with different operand types, the operator must also be provided with information on the type with which to interpret the bit patterns described by the Values' knowledge representation information.

[15]This statement is not strictly true: see below

The algorithm discussed above forces specializations *for a given construct* to be treated as mutually exclusive: If two distinct specializations for a construct are proposed during different executions of that construct, it is taken as indicating that *no* specialization is possible. Although this restriction is likely to have no major impact upon the quality of specialization performed by the system, in some cases such a constraint can inhibit the discovery of certain specializations. For an example of this phenomenon, consider a function called only once in the entire program. Suppose that this function contains a single integer *add* expression which is executed twice in the symbolic evaluation of the function (e.g., within a loop that whose body only executes twice). Suppose that in each execution of the expression, the right operand to the add expression is known to be zero. Further suppose that during the first execution the left operand to the expression is also exactly known and so that the entire add expression is recognized as evaluating to an exactly known value, while in the second case the left operand is associated with an unknown value so that no constant folding can take place. Because PARTICLE currently only permits the association of a *single* possible specialization with a given execution of a construct, for the first execution the system will opt to record only the reduction describing the possibility of entirely reducing the expression. In the second case, the only specialization possible is algebraic simplification (the replacement of the original expression by its left operand), and this specialization will be proposed. Unfortunately, although this algebraic specialization was *possible* in the context of the first execution it could not be recorded as possible. As a result, PARTICLE will note that the two proposed specializations for the expression are incompatible (because they are not the same) and conclude that *no* specialization is possible for that expression, where in fact the algebraic simplification specialization *is* applicable in all contexts. This difficulty represents a shortcoming in PARTICLE's current specialization strategy, and will be easily corrected by permitting a *set* of possible specializations to be associated with each execution of a construct within future versions of the project.

4.3.3 Statement Level Evaluation

The section above described the operation of the symbolic evaluator during expression evaluations; this section examines the process of statement evaluation, with particular emphasis on the strategies needed to handle unknown control flow.

4.3.3.1 General Behavior

While each class of statement supported by PARTICLE is associated with its own pattern of behavior, the functioning of all statements share certain universal characteristics. Some of these universals are described here.

In run time interpreters, statements map input states to output states and within PARTICLE, statements map approximations to input states to approximations to output states. In some cases, the production of the output state may be trivial (e.g., in the simulation of an assignment statement), while in other cases the creation of an output state approximation is involved (e.g., the abstraction of an loop).

Just as the the symbolic evaluator may discover opportunities for specialization during expression evaluation, optimizations may be uncovered while executing statements. For example, it may be discovered that an *if* statement is associated with a predicate which is known to be *false*, allowing the application of a specialization that eliminates the conditional and the consequent and leaves only the alternative residual. Similarly, it may be discovered that a loop predicate evaluates to *true*, permitting one iteration of the loop to be unrolled. Just as expressions were associated with a rather small set of possible specializations, the set of specializations permitted for statements is rather compact. (See figure 4-8.) As was the case for expressions, specializations for statements are represented and performed by reduction templates. (While specializations for expressions were typically discovered rather deep within the simulation of the operations for those expressions, statement-level specializations are discovered at the top level of simulation.) As described in section 4.3.2.2, reduction annotations for statements are created and treated in the same manner as annotations for expressions: An annotation to a statement is only made if it is compatible with (is associated with the same specialization as) any preexisting annotations for that statement.

In certain circumstances a path of symbolic execution may be terminated. Such termination may result from the transfer of control to a "synchronization point" (e.g., the beginning or end of a loop or a branch label) within the program being evaluated, from the invocation of a function call known to be non-terminating (either a call to a routine known to be non-terminating, such as a routine to end program execution, or a call to a recursive call determined during function abstraction to loop without end), from a recursive function call in the first step of attempting to arrive at an approximation to the return state of that function (see section 4.3.5.2.), or from a statement that has engaged in behavior known to be illegal (in which case this path of execution is judged to be speculative but incorrect). PARTICLE indicates such statement non-termination by the return of a "bottom" memory model from that statement. Return of this memory model leads to the termination of this path within the evaluator, and allows the evaluator to continue by simulating other possible paths of execution. (Note that the least upper bound of the bottom memory model with any other memory model is that other memory model: See section 4.3.3.2.1.2.)

4.3.3.2 Control Flow

The general means by which one simulates control flow during statement-level symbolic execution is simple: The symbolic evaluator evaluates each construct of the program in turn, modeling each of their effects upon the memory model; when some control construct is encountered (whether it be a conditional, loop, or goto), it is directly simulated by the symbolic evaluator. Unfortunately, patterns of control flow are frequently highly dependent on data. While the paths taken at run time can sometimes be statically determined (see below, [53]), this is by no means always the case. As a result, during symbolic evaluation there may be instances where the symbolic evaluator has incomplete information to determine what the "next" operation will be in some path execution, and can therefore not directly simulate that operation. Although some previous symbolic evaluation systems [5][49][23] have simply avoided dealing with the difficulties posed by uncertain patterns of control flow, any system aspiring to practical and general use must directly confront these complications. The PARTICLE system is currently capable of straightforwardly handling the vast majority of the control flow patterns seen in input programs; extensions to handle entirely general patterns of control flow may be added in future modifications to the project. (See section 5.2.1.) PARTICLE's mechanisms to provide high quality approximations to the effects of loop execution and function calls can permit equally dramatic improvements over traditional analyses, thereby permitting the implementation aggressive specializations beyond the range of traditional compilers. We now turn to discuss PARTICLE's method for dealing with each major control flow construct.

4.3.3.2.1 Conditionals The handling of conditionals (*if* and *if-else* combinations) within PARTICLE illustrates the fundamental ideas behind the handling of all control flow constructs, so it will warrant particularly careful explanation.

There are two very different cases to handle when dealing with control flow constructs: instances in which the control flow following with the construct is *known*, and cases in which the control flow associated with the construct is *unknown*. Each of these cases is discussed below.

4.3.3.2.1.1 Known Conditionals The handling of conditionals whose predicates evaluate to Values that are definitively known to be either *false* (zero within the language being evaluated) or definitively known to be *true* (non-zero)[16] is straightforward. In this case, we know which branch (if any) of the conditional will be taken, and simply evaluate the statement (consequent or alternative) associated with that branch.

4.3.3.2.1.2 Conditionals with Unknown Predicates The handling of cases in which the predicate of the conditional is not known to be zero and not known to be non-zero ("unknown conditionals") requires far more care than is needed for the handling of conditionals associated with known predicates ("known conditionals"). For unknown conditionals,*while the symbolic evaluator knows that exactly one of the branches of the conditional will be taken at run time, it does not know which branch this will be.* In order to conservatively model the effect of executing the conditional upon program state, the symbolic evaluator must construct a model of program state that is a conservative approximation of the state that would be produced by the execution of *either* branch of the conditional (in order for such a state to simultaneously be a legal representation of the result of executing the consequent or executing the alternative.)[17] The way in which we accomplish the creation of such a model of memory model is to first execute the consequent, yielding an approximation to the run time state that would obtain after the execution of the consequent. We then execute the alternative *starting with the original state with which we entered the conditional.* This provides use with an approximation to the run time state that would obtain after executing the alternative branch of the conditional.

At this point, we have obtained two different states, each of which captures an approximation to the run time state that would exist after the execution of one of the branches of the conditional. Since we do not know *which* path will be taken at run time, we do not know

[16]Note that a Value need not be *precisely* known to be known to be non-zero: many range and set approximations can be established as non-zero.

[17]Note that within this section we are implicitly making use of assumptions concerning the location and character of the program "join points": As discussed in section 3.2.2, there are many different points at which we could join together different speculative branches of computation spawned by uncertainties in control flow. In general, each separate fork of symbolic evaluation created at points of unknown control flow could be allowed to continue for an arbitrary length of time before being merged into a single state approximation that conservatively captures the possible effects of all paths from that fork point. While allowing long speculative execution paths may allow for more precise analysis, it also carries with it additional performance overhead (because *each* such path may execute large amounts of the same pieces of code as are executed by the other threads).

which of these states will be the appropriate conservative approximation to the actual run time state after executing this conditional (although we know that one of them will serves this role). At this point, the job of the symbolic evaluator is to create a single new state that is a legal approximation to *either* possible resulting state, and thus represents a legal approximation to the state of the program after the program has taken either branch of the conditional. We can create such an approximation to the state by taking a *least upper bound* (or $\sqcup$) of the two states produced by the separate execution of each branch of the conditional.

The least upper bound of two program states in PARTICLE is simply a state consisting of the $\sqcup$ of the *contents of each program resource* (memory locations, stack pointer, etc.)[18] for each of these states within the pointed complete partial ordering associated with the Value domain. Recall, however, that while the atomic elements in the abstract domains in PARTICLE are *Values*, individual memory locations in PARTICLE do not store complete Values – instead, they store Byte Clipped Values. (See section 4.2.2.3.) In order to take the $\sqcup$ of two Byte Clipped Values, we must incorporate these Byte Clipped Values into the same byte position in two Values, and take the $\sqcup$ of these two Values. In the Value produced by the $\sqcup$ of these two Values will be of the same size as the two operands to the $\sqcup$, and will thus have a Byte Clipped Value at the same position as the original byte clipped values occupied in the operands to the $\sqcup$. Because the Value produced by the $\sqcup$ operation is a valid approximation to either of the operands to the $\sqcup$, any subpart of this Value is a valid approximation to either associated subpart of the operands of that $\sqcup$ operation. (Intuitively, the result of the $\sqcup$ must be a valid general approximation to the two arguments to the $\sqcup$ in all respects (i.e. the result of the $\sqcup$ must include the two arguments to the $\sqcup$ as "special cases") – and thus a *subpart* of that result must represent a valid approximation to the corresponding subparts of the two arguments to the $\sqcup$.) Thus, the Byte Clipped Value at the appropriate position in the result of the $\sqcup$ is guaranteed to represent a valid approximation to both of the original Byte Clipped Values. Thus, while the $\sqcup$ of two states in PARTICLE conceptually takes the $\sqcup$ s of the *contents of locations* of those states (as well as the $\sqcup$ of any other program resources), for each such location, this $\sqcup$ is *performed* by taking the $\sqcup$ s on complete Values incorporating the Byte Clipped Values.

In the discussion above, we noted that taking the $\sqcup$ of two Values would automatically take the $\sqcup$ of any particular subpart of each of the two Values (e.g., "the first byte in each such Value"). This holds true *no matter what Values* in to which we choose to incorporate the Byte Clipped Values. (In other words, this property is true of *any* Values incorporating the Byte Clipped Values as a subpart.) In particular, when creating a Value to incorporate the two Byte Clipped Values that came from each location being generalized, we have a infinitely large range of possible Values to choose from (in other words, an infinitely large range of other Byte Clipped Values to incorporate into the other positions in the Value). While the most straightforward solution might be to simply create a Value of a single byte size for each Byte Clipped Value and to simply generalize these Values, it is typically more efficient to take advantage of locality of reference (and the fact that the generalization machinery is geared to work on Values) and to try to generalize the entire Value that was written along with this Byte Clipped Value (particularly since this Value is normally just available in a cache translating Byte Clipped Value to Values:

[18]Note that all $\sqcup$ s within PARTICLE are taken between states with the same stack pointer, so that the result of the $\sqcup$ never has the potentially disastrous effect of yielding a state with an inexactly known stack pointer. This convention is enforced even within the abstraction of recursive function calls.

See section 4.2.2.3.), and thus does not involve the overhead of creating an entirely new Value abstraction. In this manner, with the application of a single Value-wise $\sqcup$ we can simultaneously generalize the Byte Clipped Value associated with the particular location to be generalized and also generalize several other Byte Clipped Values associated with neighboring locations which would likely also have to be generalized as well (because writes to memory often occur in adjacent locations, both because of the multi-byte character of most program data types and because of spatial locality of usage).

Taking the least upper bound for two states thus conceptually involves the creation of a state that is a conservative approximation to each of the two operand states on a location-by-location basis (or, more generally, on a resource-by-resource basis). This state, in turn, is generated by the application of the $\sqcup$ to the contents of each corresponding location in such states. In practice, we can take advantage of two observations made in section 4.2.3.1: The fact that the $\sqcup$ of two equivalent values is the same value, and the empirical observation that although program states can be very large, the amount of change witnessed between states is often quite small and localized. If by combining the implications of these observations we can avoid applying the $\sqcup$ to each location when generalizing two states and instead apply it to just those locations that that have possibly changed between the two states, the symbolic evaluator will be able to save an enormous amount of computational effort. By making use of the record of changes between two successive memory models (see section 4.2.3.1.), we can do exactly that. In particular, to find the maximal set of possible changed locations between two arbitrary program states, we need only collect the set of changed locations along the paths from each memory model to their nearest common ancestor. The union of these sets of locations will yield the maximal set of locations whose contents are possibly different between the two states. By basing the result of the $\sqcup$ on one of the two states whose $\sqcup$ is being taken and taking the $\sqcup$ of only the locations whose contents may have changed, the symbolic evaluator can rapidly create a new state approximation representing the $\sqcup$ of the two given states.

We can thus create an approximation to the output run time state of a conditional with an unknown predicate value by separately executing each branch of the conditional and taking the least upper bound of the resulting states by taking a location-wise $\sqcup$ for the contents of those states.

4.3.3.2.2 Loops The desugaring system discussed in section 3.5.4 canonicalizes all loop constructs, so the comments here can be confined to the handling of a single loop form: A loop with an initial test and body. Such loops can also be associated with *continue* and *break* statements (to transfer control to the next iteration of the loop or to the end of the loop), and the handling of each of these constructs will be discussed in below.

As noted in section 3.4, PARTICLE provides the user with a flexible termination policy for the symbolic evaluator. This section briefly reviews how termination is enforced for loops. Pending the invention of a safe but less restrictive mechanism for guaranteeing termination of the symbolic evaluation (see section 5.3.2.1.2.), PARTICLE employs a very crude method for ensuring termination: The maintainance of explicit bounds on the number of iterations a loop can execute before it is "abstracted" to produce a more conservative approximation to arbitrarily many iterations of that loop within a finite time. (See section 4.3.3.2.2.3.) As discussed in section 3.4, PARTICLE maintains separate bounds for bounding the number of iterations possible in loops associated with *known* and *unknown* iteration conditions; the idea here is that every iteration simulated for a loop with *known* iteration conditions is known to correspond to an iteration of that loop at run time, while a simulation of the body of a loop with unknown iteration conditions may or may not correspond to an actual iteration of that loop at run time. It may be that while we are willing to pay the performance cost of allowing many symbolic evaluation of loops that are guaranteed to run at run time, we will wish to devote far fewer computational resources to the simulation of the effects of loop iterations that may not be executed at run time; the separate specification of maximal iteration counts for known and unknown loop iteration conditions permits the capturing of a crude approximation to this difference in preferences.[19] In accordance with the philosophy of allowing the user maximal influence over important decisions during symbolic evaluation, PARTICLE also provides a mechanism for specifying maximal known or unknown iteration iteration counts for a particular loop in the source program. Like many fine-grained user directions, this mechanism is performed through the general mechanism of external calls from within the user program (see section 4.3.5.1.2.), and can thus be implemented within a fully dynamic manner (i.e., if desired, the settings can be computed within the symbolic evaluation of the program itself). While the coarse-grained mechanism allows the specification of default loop conditions for the entire program, this form of specification provides the user with the option of widely varying such preferences during program execution and adapting the quality of the symbolic evaluation to the low-level structure and character of the particular program being evaluated.

4.3.3.2.2.1 Handling Loops with Known Iteration Conditions In section 4.3.3.2.1.1 we noted that the handling of conditionals whose predicate evaluates to a known truth value is straightforward; the handling of loops where the truth value of the iteration condition is known *for all iterations* is equally straightforward and mirrors the process of loop evaluation in traditional interpreters. At the beginning of each loop iteration, PARTICLE evaluates the iteration condition in the current state. If the iteration condition is *true*, the loop body is executed (starting with the current state). If the iteration condition is *false*, the loop is terminated and we continue on to the next statement.

This simple picture is slightly complicated by the need to handle *continue* and *break* constructs. At run time (or in a language interpreter) encountering such a statement within a loop body requires that we branch immediately to the implied destination: the beginning or end of the loop. During symbolic evaluation we may not be able to do this. In particular, in keeping with its practice of keeping program control flow simulation join points as close as possible to the corresponding fork point, PARTICLE simulates loops on an iteration-by-iteration basis, resulting in a system in which all speculative paths of execution within a given iteration of the loop are brought to completion before beginning the next iteration or leaving the loop. It may be (and perhaps is likely to be) that when we encounter a "break" or "continue" statement we are inside an *unknown conditional* within the loop body, so that there are many other paths of computation within this iteration which we must finish processing before beginning the processing of the next loop iteration or leaving the loop altogether. For example, when we encounter a *break* statement, we may be inside an unknown conditional, the other branch of which actually continues on to the top of the loop – it would be inaccurate and (worse) *unsafe* to simply proceed immediately to the end of the loop – for under *some* run time conditions the

[19]For a more complete critique of the current system, see sections 3.4 and 5.3.2.1.2.

loop execution may continue, and failure to simulate such iterations could lead to inaccurate approximations to the run time behavior of the loop.

In order to handle *continue* and *break* statements without simply beginning simulation at the appropriate destination, we will make use of a technique that represents a straightforward generalization of the handling unknown control flow in conditionals and that anticipates the handling of more general control flow constructs (such as *goto* statements): The annotation of program points with "pending state." When making use of this technique, we do not proceed directly to the construct to which control is branching. Instead, we record our current state (state at the time that the *continue* or *break* statement is encountered) as being one of the possible states associated with the target program point (intuitively, this tells the target point that the current state is one of the possible states with which it could be reached), and terminate the current path of execution (intuitively, to allow other pending paths of symbolic evaluation to proceed). The enclosing construct (the *while* loop in the case of *break* and *continue*, or the function in the case of a general *goto*) will later place the current instruction pointer for the symbolic evaluator at the destination point (after all other pending branches of computation have finished), at which time we adopt an approximation to all of the set of states that could reach that point[20] and continue executing. In the case of known loops, each iteration begins with the execution of the generalization set of states that could have reached the top of the loop during the last iteration (whether through the execution of a *continue* or through reaching the end of loop body). Similarly, the "end of loop" state accumulates generalizations of possible exit states (states associated with *break* statements) until the loop is terminated, at which point we exit the construct with a state that represents a generalization of all states that could reach the end of the loop.

4.3.3.2.2.2 Handling Loops with Unknown Iteration Conditions The treatment of loops with unknown termination conditions represents a straightforward generalization of the mechanisms for the handling of loops with known iteration conditions. The same set of mechanisms is used, and the general algorithms for executing loop bodies and the handling of *continue* and *break* statements is the same. The sole difference in handling of the two constructs concerns the fact that before executing *each* new iteration of the loop associated with a loop iteration condition that evaluates to an unknown boolean value, we must annotate the post-state of the loop with the current state to indicate that is represents a possible loop termination state. It is to be expected that the user will likely insist upon performing relatively few loop iterations with unknown predicates prior to deciding to commit to the process of "abstraction", which begins the construction of a safe approximation to the effects of *any number of loop iterations* within a finite period of time.[21]

[20]In practice, this approximation to all of the possible states associated with that target point is constructed incrementally during each annotation through successive generalization of any existing annotation with each new possible state.

[21]Although beginning the process of abstraction is frequently desirable immediately upon encountering a loop, there are some unusual but not entirely uncommon situations under which it is desirable to simulate the loop on a more fine-grained basis for several iterations prior to abstracting, centering upon the fact that there may be initial transients after which the character of the loop changes and/or the loop termination conditions may become known. (e.g. Consider a loop in the program being symbolically evaluated which iterates some successive relaxation algorithm for a fixed number of steps or until it reaches convergence, whichever comes first. In such

4.3.3.2.2.3 The Process of Loop Abstraction When PARTICLE detects a loop that has exceeded the maximal number of iterations specified as permissible by the user, it invokes a process that is guaranteed to create an approximation to the state following the loop within a finite amount of time, regardless of how long the loop will execute at run time (possibly forever). The process of abstraction in loops is relatively straightforward (and can be thought of as similar to the process of abstracting tail-recursive functions in the presence of recursion: See section 4.3.5.2.). Conceptually, we wish to create an approximation to the output state of the loop that holds true for *arbitrarily many iterations of the loop.* Almost by definition, any such approximation will have the property that executing the loop an additional time upon the approximation will yield *the same approximation*, for by hypothesis the approximation describes the result of the loop execution for *any* number of iterations. Thus, the general strategy for abstracting a loop is clear: We will iterate the loop and generalize the approximation to the state after each such iteration to produce a new legal memory model that simultaneous represents an approximation to the results of all iterations we have already executed. We continue generalizing this approximation, until that approximation no longer changes after an iteration of the loop. At this point, we have already created an approximation to the output state for all iterations from the first to the current iteration (since we have been generalizing together this output state for each iteration). Moreover, since the output state approximation no longer changes under execution of the loop body, we know that the current approximation is a valid approximation to the effects of all further possible loop iterations. (It is certainly an approximation for the output of the *next* iteration, since executing the loop body leaves the output approximation unchanged. And it is certainly an approximation for the iteration after *that* since it is an approximation to the output of the next iteration, and executing one more iteration after that will leave it unchanged...) Thus, a valid approximation to the effect of *any number of iterations of the loop* can be found by the generalization of the original state upon entering the loop (since it may be that *no* loop iterations will be executed at run time) with this, an approximation to the output state for executing one or more iterations of the loop.[22]

The process of fixed-point iteration lies at the core of the process of creating a legal approximation to the effects of arbitrarily many iterations of a loop. In the description above, we have assumed two things: that there exists a least fixed point for the abstraction process and that the process above will find it within finite time. *Assuming bounded memory size*, the existence of a least fixed point for the loop evaluation process can be shown by demonstrating that the abstract state domains employed by PARTICLE represent pointed complete partial orders and that the process of successive loop iteration and abstraction represents a continuous function [48] The fact that PARTICLE's memory model domains represent a pointed cpo (in fact, a lattice) can be seen by reducing the problem to one of demonstrating that the Value domain represents a pointed cpo. The fact that this condition obtains can be informally seen by considering the "exactly known" concrete Values as simply 1-tuples of the set representation,

a situation, careful iteration-by-iteration simulation in the presence of uncertainty concerning loop behavior will still eventually allow definitive of loop termination – even if convergence cannot be demonstrated, eventually the loop will iterate the appointed number of times and will terminate.)

[22]Note that the discussion above has largely appealed to a simplistic dichotomy between loops with known loop termination conditions and those with unknown termination conditions. In practice, many loops will have termination conditions that may alternate between these two conditions; such loops can be handled by a straightforward combination of the above methods.

using the 0-tuple of the set representation as $\perp$, and considering the range representations as "overlying" the set elements in the domain. Finally, a totally unknown Value represents the $\top$ element in the lattice. The overall structure of the Value pointed cpo is depicted in figure 4-10. A proof that the loop abstraction mechanism represents a continuous function is not included here.

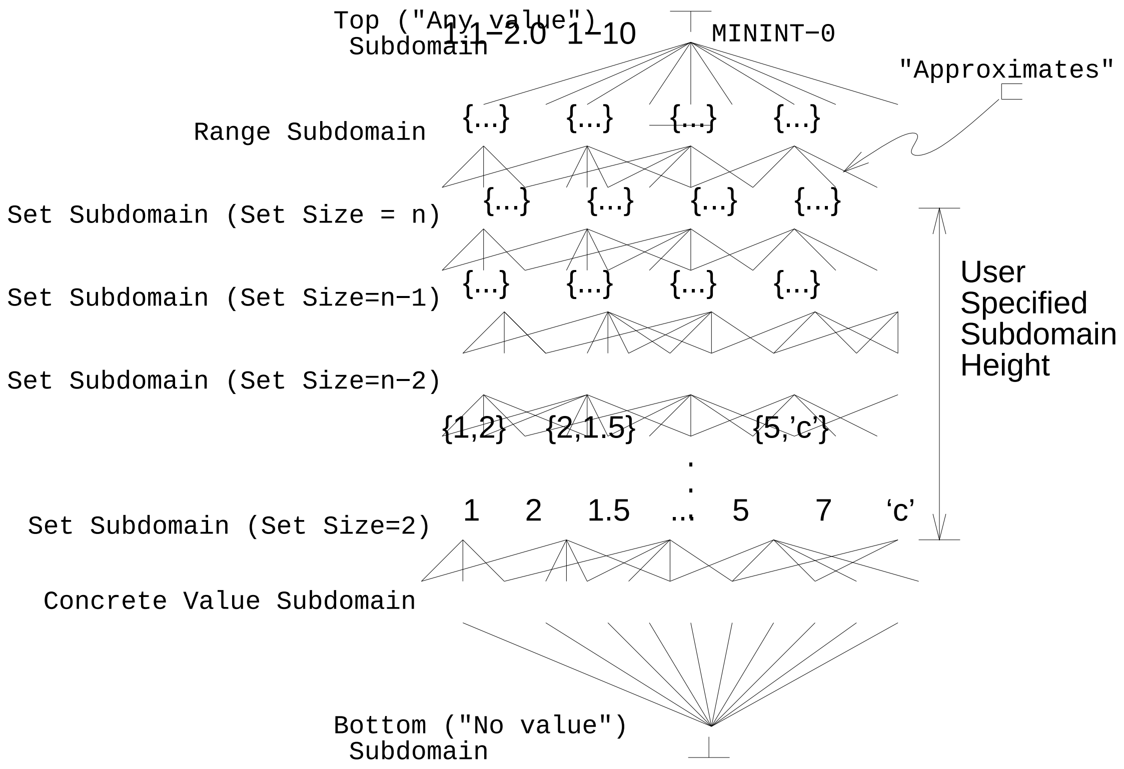
Top ("Any value") Subdomain
1.1–2.0
1–10
MININT–0
"Approximates"
Range Subdomain
{...}
Set Subdomain (Set Size = n)
Set Subdomain (Set Size=n–1)
Set Subdomain (Set Size=n–2)
{1,2}
{2,1.5}
{5,'c'}
Set Subdomain (Set Size=2)
1
2
1.5
...
5
7
'c'
Concrete Value Subdomain
User Specified Subdomain Height
Bottom ("No value") Subdomain

Figure 4-10: The diagram illustrates the basic structure of PARTICLE's value lattice. While the height of the range subdomain is currently fixed to one, the height of the set subdomain is set by the user.

The fact that the process above will eventually attain *some* fixed point is guaranteed by the finite height of the lattice upon which abstraction is performed. In particular, the process of abstraction involves the creation of a memory model where the contents of each location is associated with the least upper bounds for the contents of the locations in the memory models being abstracted. PARTICLE requires (finite) bounds on the maximum sizes of its set representations for Values. These bounds limit the height of the associated lattice (formed by set inclusion for Values). The height of the "range" lattice is implicitly bounded by the size of (always finite) domain for the types used for the endpoints. The only additional elements in the abstract Value lattice are the top element and the the concrete Values. As a result, abstraction of the contents of a given location in the memory model is guaranteed to terminate within a bounded amount of time. Because there exist only finitely many locations in the memory model, the abstraction the contents of every location to the associated least fixed point Value is ensured termination within a finite amount of time. By starting with appropriate initial conditions (essentially, we start with "bottom" memory model as the default loop output representation, before generalizing it with any loop termination states), we can guarantee that the fixed point we will reach will be the *least* such fixed point.

Having discussed the process of loop abstraction, it is worth pausing to consider what information normally captured by the iteration-by-iteration symbolic evaluation of a loop is lost by following this procedure. Since the abstraction process in essence involves the simulation of arbitrarily many loop iterations within a finite period of time, it is to be expected that the approximations to the effects of the loop obtained through this mechanism would be greatly inferior to those obtained through the separate simulation of each loop iteration. By looking a bit more carefully at the form the degradation takes, we can hope to discover from whence it arises and learn to recognize cases where abstraction will be particularly damaging to program analysis and take appropriate measures to prevent premature abstraction in such cases. The fundamental effect of abstraction upon loop analysis is a "mixing together" the models of the states and Values associated with several different loop iterations: If we are executing a loop in an iteration-by-iteration manner, the states for a given iteration capture an isochronous "slice" of the program's values. On the other hand, during the process of abstraction we are generalizing together and operating within states that contain Values associated with many different possible points in the execution of the program. At the end of the loop, each Value is associated with the Values it maintains across many loop iterations (rather than with the Value it would be associated with the (likely much smaller set of) *particular states* that would actually obtain when the loop terminates at run time). This loss of knowledge of the Values of program quantities can prevent effective program specialization. (See figure 4-11.)

Although during abstraction we can lose substantial information about the Values associated with particular program quantities, often the effects of this loss of information are greatly compounded by the total loss of information about the *simultaneous existence* of Values. (See section 3.2.2.) By combining together memory models from many different points, abstraction makes it impossible to keep track of which different Values are simultaneously in existence within a given memory state that during loop abstraction. (This tremendous "mixing together" of memory models can be recognized as an unavoidable consequence of abstraction, when it is recognized that each iteration during abstraction may serve to model the effects of *arbitrarily many* iterations at run time. As a result, the memory models manipulated during abstraction must capture information about possible program states from throughout many points in the execution of the loop.) Figure 4-12 provides an example of where knowledge of "simultaneity" of Values can be extremely important. Note that in all such cases, the deleterious effects of loops on program analysis can be magnified by the presence of code that is extremely sensitive to slight inaccuracies in analysis (e.g., array or pointer code). The example in figure 4-11 illustrates this phenomenon.

4.3.4 Gotos

The general manner in which *goto* statements are handled during symbolic evaluation is suggested by the mechanisms used to support "controlled goto" statements such as *continue* and *break*. First we annotate the target location with the current state (indicating that the current state is one of the possible states that can reach that point). We then terminate the current

```
for (i = 0; i < 256; i++)
        {
        array[i] = i;
        }

 /*
   While fine-grained loop simulation will be able to exactly
   simulate this loop initialization, far less information is
   available after abstraction of the loop effects.  In
   particular, the simulated program state at the end of the
   loop will have both i and all element values of the array
   associated with at least the range of possible values 0-255.

  */
```

Figure 4-11: The Abstraction of program loops can prevent PARTICLE from recognizing and exploiting a great deal of static information concerning program behavior. Although fine-grained analysis can precisely derive the effects of the loop shown, the use of abstraction to accelerate such analysis will yield a grossly conservative approximation to such effects.

path of termination, and rely on the higher-level execution mechanisms to eventually bring us to the target point. In the case of a loop, after processing a *continue* statement we rely on the process controlling the symbolic evaluation of the loop to place us at the beginning of the next iteration. Similarly, a *break* statement terminates the current path of execution and relies on higher-level mechanisms to eventually place the control flow at end of the loop and allow execution to continue from that point. For a *goto*, the symbolic evaluation of the enclosing function is responsible for guaranteeing that control flow will eventually reach or be placed at any label associated with a pending thread of control.

Just as there exist loops in control flow due to *while* loops and recursion, there loops can exist loops in which a *goto* is used as the back edge. Since we wish to be able to enforce timely termination of the symbolic execution of *goto* loops as we enforce the termination of other loop types, we must augment the strategy above to provide some mechanism for detecting "excessive" looping. Such a mechanism can be created by associating each possible *goto* target (marked by a label) with a data structure parameterized by call and *goto* statement recording the number of times that we have reached this label from the named *goto* statement during that call. (The parameterization by call is needed to correctly perform this process in the presence of recursion.) Whenever we reach a label, we check to see if we have exceeded the limit on the number of possible *goto*s from a certain statement (just as we check to see if too many iterations have taken place within a loop when judging whether it is desirable to begin the

```
int strcmp(char *sz1, char *sz2)
{

  for(; *sz1 == *sz2 ;  sz1++, sz2++)
    {
    /*
        During abstraction of this loop, knowledge of which
        values of sz1 and sz2 exist simultaneously is lost.
        This prevents the recognition (and exploitation) of cases where
        the strings compare exactly.
     */


    if (*sz1 == '\0')
        {
        return(0);
        }
    }

  if (*sz1 > *sz2)
    return(1);
  else
    return(-1);
 }
```

Figure 4-12: Each simulated state during loop abstraction mixes together the states associated with many different iterations of the loop at run time, and the system loses knowledge about which values exist simultaneously in program states. As a result, operations comparing different quantities in such loops will typically not be able to be evaluated at compile time. In such cases, even a simple string comparison routine evaluated on statically known strings will not be capable of being reduced.

abstraction process). If we determine that one of the *goto* statements has indeed exhausted its quota, we must generalize the loop between the target statement and that *goto*. For the general case (which can involve “overlapping loops”, as illustrated figure 4-13.), the process of abstracting within PARTICLE's framework is extremely awkward, involving an interleaving of the abstraction of any loops surrounding the target statement and the overlapping loop.[23]

[23]Even for loops that where the target is located inside a statement in which the *goto* is itself not located, the mechanics of abstraction within PARTICLE are rather baroque due to the implementation of evaluation on top of an abstract-syntax representation for the program (see section 4.2.1.). Conceptually, however, the process can be carried out as described below.

If there are no overlapping loops present, however, the process is much simpler: We simply execute the code between the current statement and the errant *goto* statement, abstracting as described in section 4.3.3.2.2.3. It is important to emphasize that this process is only used in instances where *goto* constitutes the back-edge of a loop in control flow.

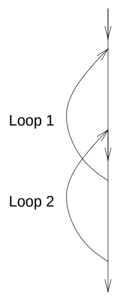

Figure 4-13: The abstraction of overlapping loops is awkward within the abstract interpretation framework used by PARTICLE: It involves interleaving between iterations of each separate loop.

PARTICLE currently does not handle the abstraction of goto-based loops, although construction of a mechanism to handle this would be relatively straightforward.

4.3.5 Function Calls

4.3.5.1 General Handling

4.3.5.1.1 The Function Call Process In the absence of recursion, the simulation of function calls within PARTICLE is straightforward and mirrors common techniques for implementing function calls at run time: Space is allocated for the return value of the called function, actual parameters to the function are evaluated and pushed onto the stack[24], the return continuation is saved, and control is transferred to the target function. Following the complete execution of the called function, the activation record for the function call is popped from the stack and control transfers back to the caller.[25]

4.3.5.1.2 Calls to External Functions Although PARTICLE ideally operates on programs in which as much source code as possible is available for simulation by the system, the framework supports calls to external routines whose bodies are not themselves available for symbolic evaluation (e.g., library routines or external modules of a large system). Within the current implementation of PARTICLE, the effects of external routines on program state are approximated by stubs provided by the user and linked into PARTICLE at compile-compile time

[24]Note that in order to efficiently and cleanly handle functions accepting variable number of arguments, the values of the actual parameters are pushed onto the stack in *reverse* order, allowing the first argument to always appear at some known offset from the frame pointer.

[25]Note that like the points immediately after loops and conditionals, the code to terminate a function execution represents a join point for symbolic evaluation: All paths of symbolic evaluation spawned in the course of function execution must synchronize at this point and be generalized into a general return state approximation prior to function return.

(i.e., at the time at which PARTICLE itself is compiled). The use of general-purpose stubs for capturing the behavior of external calls offers a natural and simple manner of achieving three important benefits: The precise specification and circumscription of the possible effects of functions whose bodies are not available for analysis, the creation of flexible pragmas allowing the user to dynamically and adaptively control the and quality and timing of program analysis during symbolic evaluation, and the ability to seamlessly permit specialization of an entire program with respect to external input. The benefits offered by the external call interface in each of these areas are briefly described below.

The ability to implement general-purpose function stubs avoids the need to make uniform and grossly conservative assumptions about the effects of external routines and allows the user the option of creating stubs offering precise modeling of the effects of particular calls to those routines. Because the stubs for external routines run within the PARTICLE framework, they have full access to the symbolic evaluator's model of program state at the time of the call and to the full range of knowledge collected by the system about program quantities. This information permits the exact functionality of the external routine to be simulated wherever desirable and possible (e.g., allowing the performance of a string copy or data sorting if the source data is completely known), and frequently permits highly precise circumscription of the effects of the call even where precise modeling of target function behavior is not possible. (e.g., Where the contents of the source string or array to be sorted are not available, a stub might capture the possible effects of the call by simply invalidating or partially invalidating any knowledge of just that area of memory.) Even simple stubs can frequently capture vastly more precise information about the effects of an external call than is possible to express in existing systems. The flexibility and specificity of stubs can prove important when compiling certain classes of code: in existing compilers, typically the compiler is forced to make the grossly conservative assumption that any particular external call may use and modify all global variables and overwrite the referents of any available pointers. In certain cases, use of such assumptions can have seriously negative consequences for the quality of a compiler's optimizations. By permitting fine-grained modeling of the possible effects of external calls, stubs can safely capture vastly more static information about program behavior than is possible with traditional techniques.

Earlier sections have described the wide variety of "knobs" PARTICLE offers to the user that allow fine grained control over the character and quality of the symbolic evaluation performed. For most of the specifications influencing specializer behavior, PARTICLE allows the user to exercise such control in an extremely fine grained manner by means of calls to stubs made during the process of symbolic evaluation. Although it seems unlikely that the user will routinely find it necessary to use of the full power and generality of such specification mechanisms, under certain conditions such control may be desirable: For example, the user could insert code within a program loop to successively and dynamically restrict the quality of program analysis over the course of the symbolic evaluation of the as loop. Such adaptive control over the precision of program specification could permit highly precise analysis to be performed without tremendous compile time overhead, and is easily implemented through the use of stub calls. (Automatic means for performing such adaptive control are also of clear interest: See section 5.3.2.1.) In the common case, however, it is expected that the fine-grained specification enabled by the calls to external routines during symbolic evaluation is likely to be exploited in a far less general manner, and frequently in a purely static way for a given call site.[26] External stubs provide a straightforward and convenient manner of specifying either simple or sophisticated desires concerning the quality of program analysis provided by the system.

This document has emphasized the performance advantages arising from specialization of a program with respect to external program input. The creation of of stubs to model external input routines provides a simple but general means by which to permit this form of specialization. In particular, all external input into C programs is typically performed through the use of calls to external functions (such as the *fgetc*, *fgets* or even *scanf* of the C standard I/O library). By simply providing appropriate stubs for such routines within PARTICLE, the full functionality of the original input functions can be trivially provided where possible (by simply performing any call during symbolic evaluation that has sufficient static information by means of a direct call to the corresponding run time library routine or other external function). In addition, the behavior of such input functions can be supplemented so as to permit the specification of incompletely known input within the structure of PARTICLE's abstract domains. For example, a numerical input routine (or the numerical component of a general input routine such as *scanf*) could allow the user to specify a piece of data as either an exactly known value, a member of some small set of values, a completely unknown value, or as an element of some range of possible numerical values. Similarly, a string or character input routine might allow the user to phrase the input string or character in some syntax permitting the specification of the first three of these categories.[27] The implementation of stubs related to external input differs in no significant manner from that of stubs simulating other external functions. The external function stubs within PARTICLE provides a clean, simple, and general purpose mechanism by which external input to a program can be specified fully, partially, or left completely unspecified.

[26]For example, the user may wish to specify a fixed maximal number of iterations permitted prior to abstraction or the richness of the abstract domains employed by PARTICLE on a per-loop basis, rather than simply specifying criteria for the program as a whole. Such a specification could simply be made with a call to a stub routine with some fixed argument.

[27]Even fairly sophisticated functionality can be implemented very simply within this framework. For example, many important low-level library routines manipulate data which changes between invocations of a given program and to which a program can therefore not be legally specialized. In particular, some of the values maintained at compilation time (such as file I/O information created by an external call to the *fopen* function) are frequently quantities dependent on system state whose particular values may change between different calls (invocations) of the program as a whole. Although it is not in general possible to know at compile time the exact run time values that will be associated system state dependent quantities, certain critically important classes of *operations* on these values *will execute in equivalent manners* between invocations of the program. For instance, an I/O operation opening a file may create system-state related information that differs between different run time invocations of the program (e.g., because there happens to be a different number of files open at these different times, the file handles associated with the opening of a *particular file* with unchanged contents may be different between different executions of the program), but certain operations that operate on this information (e.g., operations that read the next line) should act in exactly the same manner whether at run or partial evaluation time. Thus, while the program may not be legally specialized with respect to a precise file handle value associated with some file I/O information, a program specializer should support the ability to *read from* a file using that file handle at compile time and permit specialization with respect to the information so read – *despite the fact that the run time value of the file handle itself is not known.* Although the proper support of such system state dependent quantities (which share many characteristics of "unknown" values but are still capable of being operated upon in well-defined manners) requires the use of some additional machinery within the stubs, the stub framework provided by PARTICLE provides a means by which this machinery can be simply and cleanly implemented.

4.3.5.2 Abstraction of Recursive Applications

4.3.5.2.1 General Techniques In accordance with the same philosophy that enforces the timely termination of program loops, PARTICLE maintains flexible but strict mechanisms to guarantee the termination of recursive function calls. For loops, PARTICLE recognized a distinction between loops where the iteration conditions were known to be true during symbolic evaluation (cases in which case each iteration being simulated was known to reflect an actual iteration of the loop at run time, providing that the loop itself were reached), and loops in which the truth value of the iteration conditions was not known during symbolic evaluation (and where it is therefore uncertain whether a particular loop iteration simulated during symbolic evaluation will actually be matched by a corresponding iteration at run time). PARTICLE allowed the user to formulate distinct rules for handling each of the two types of loop. Similar distinctions and provisions are made for run time loops in control flow that are effected by recursion: PARTICLE distinguishes between recursive sequences in which it is *known* that the recursion will take place at run time (provided that control reaches the initial call) and sequences in which the recursion being simulated during symbolic evaluation is merely speculative. The user is allowed to specify different limits on the maximal depth of recursion to permit under each of these conditions, before the process of loop abstraction is begun. As was the case for program loops, the user can specify the parameters relating to the precision of simulation on both a coarse-grained level (for the entire file) and in a fine-grained manner (via specifications for these parameters that can be set dynamically during symbolic evaluation through the mechanism of external calls, allowing the user arbitrarily precise and adaptive control over the quality of symbolic evaluation within particular sequences of recursive calls).

Once a recursive calling sequence has exceeded the maximum permissible recursion limit set by the user, a process of abstraction begins. This process is directly analogous to the process of abstraction in loops, and its purpose is to create an approximation to the effects of performing arbitrarily many recursive invocations of a function within a finite time. While the use of abstraction can greatly accelerate symbolic evaluation of a program, the use of abstraction to approximate the effects of recursive calls brings with it all of the negative effects of abstraction on the precision of program analysis. (See section 4.3.3.2.2.3.)

The general method used to abstract recursive function calls is similar to that used to perform loop abstraction; the abstraction of looping constructs can be thought of as a special instance of the abstraction of recursive function calls in which the recursive functions are always tail-recursive. The abstraction procedure proceeds in two major steps: In the first phase of abstraction, the function is called recursively, and at each point the function is entered the entry state is generalized with previously recorded entry states. This process continues until a point is reached at which the entry state approximation remains the same for each successive recursive application. This entry state approximations has been created by repeated generalization of the entry states associated with earlier recursive function calls, and is therefore guaranteed to represent a valid approximation to the entry states of such calls. Moreover, because we have reached a fixed point we know that no matter how many additional times the function is recursively invoked in this chain of invocations, the current state represents a legal approximation to the entry state for every such invocation. (See figure 4-14.) Thus, when entry state abstraction reaches a fixed point the current entry state represents a valid approximation to *all* possible entry states of the function throughout the recursive sequence being simulated.

Given that the symbolic evaluation has obtained a legal approximation to the entry state

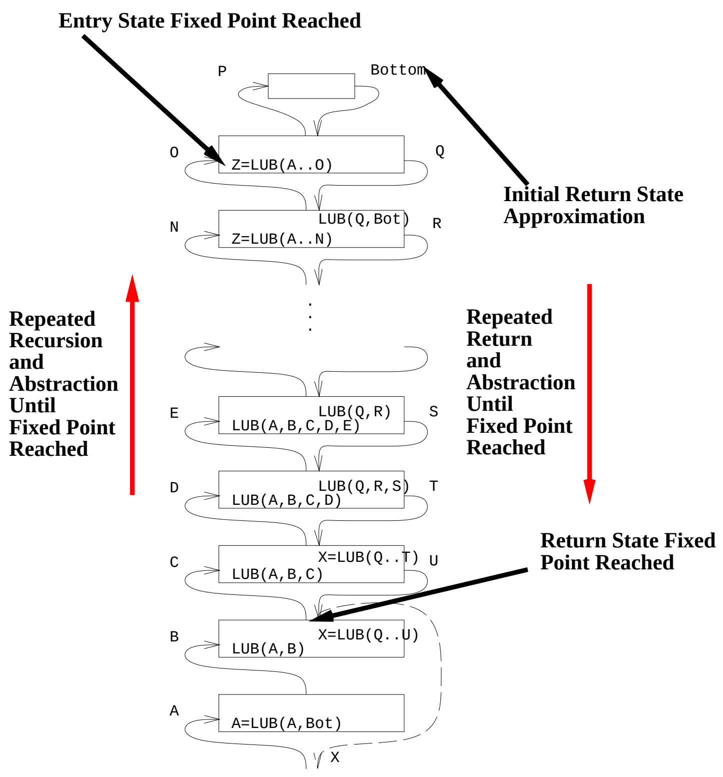
Entry State Fixed Point Reached
P
Bottom
O
Z=LUB(A..O)
Q
Initial Return State Approximation
N
LUB(Q,Bot)
Z=LUB(A..N)
R
Repeated Recursion and Abstraction Until Fixed Point Reached
Repeated Return and Abstraction Until Fixed Point Reached
E
LUB(Q,R)
LUB(A,B,C,D,E)
S
D
LUB(Q,R,S)
LUB(A,B,C,D)
T
C
X=LUB(Q..T)
LUB(A,B,C)
U
Return State Fixed Point Reached
B
X=LUB(Q..U)
LUB(A,B)
A
A=LUB(A,Bot)
X

Figure 4-14: The figure above illustrates the process by which PARTICLE creates an approximation to the effects of arbitrarily many recursive invocations of a function within a finite amount of time. The process of constructing such an approximation takes place in two major phases of fixed-point iteration: The first phase constructs an approximation to the entry state to the recursive function after any number of iteratiosn, while the second stage builds up a model of the exit state for that function at any point in the recursive sequence.

of the function after any number of recursive invocations, it may appear possible to simply evaluate the body of the function and note any legal specializations discovered. Unfortunately, one more piece of information is required in order to allow symbolic evaluation of the function body to proceed beyond the recursive call: A legal approximation to the *return* state of the function after any number of recursive invocations. (Note that this step is not required in the abstraction of loops, because they are isomorphic to tail-recursive functions; the abstraction of loops requires merely the abstraction of the loop entry state.)

The approximation to the return state is created through a relaxation algorithm very similar to the ones employed above. The approximation to the *entry* state of the recursive call was created by the abstraction of each successive entry state generate possible during symbolic evaluation of the recursively called function, starting with an approximation to the function entry state of $\perp$. (The generalization of $\perp$ with any other state yields the other state.) This process gradually enlarged the set of all approximated entry states until a fixed point was reached. The algorithm to calculate return state works in a similar manner: Initially, the approximation to the return state (including the return *value*) of the recursive function call is $\perp$. For each step of the return state abstraction process, the abstraction mechanism simulates the effect of returning the current return state approximation from a recursive invocation of the function and continues symbolic evaluation to discover the return state of the *surrounding* function invocation. The return state of this surrounding invocation is abstracted with previous return state approximations, and the result is used as a new approximation to the return state of the internally invoked function. This process continues until a fixed point is reached, at which point the system recognizes that the current return state approximation is valid no matter how many additional function returns are processed, and thus represents a legal approximation to the return state from an *arbitrary* number of recursive function calls. Thus, the abstraction process for return states is performed by repeatedly returning successive approximations to the return and abstracting each such return state until a fixed point is reached. (See figure 4-14.)

Following the computation of a general approximation to the function return state, the system can symbolically evaluate the entire function body and safely make note of possible specializations discovered with the confidence that such specializations are valid for *any* invocation of the function within the recursive function sequence being simulated.

4.3.5.2.2 Shortcomings of the Current Abstraction Mechanisms Before turning to consider the algorithms employed by PARTICLE to make high-level specialization decisions, it is worth pausing to consider some defects of PARTICLE's current strategy for handling the abstraction of recursive function calls. In order to allow the abstraction of function entry and return states, PARTICLE currently makes use of certain assumptions about program behavior that are somewhat restrictive. The origin of these restrictions is detailed below.

In the process of constructing approximations to function entry and return states each successive abstraction takes place between states with differing sizes for the activation record stack. If abstraction of the stack contents were truly to be performed on a location-by-location basis, it is likely that no fixed point would ever be reached. For example, during the process of entry state abstraction each successive recursive call would be associated with a current activation record at different points on the activation stack. Since this stack grows monotonically with each recursive invocation of the function, each successive entry state approximation will exhibit *some* change from the previous entry state. In particular, in all non-degenerate cases the area of the stack associated with the activation record corresponding to the most recent recursive call will differ from the contents of that area in the state approximating the entry state for all previous calls, and the process of abstracting all previously encountered procedure entry states will never reach a fixed point.

In order to avoid the difficulties associated with location-by-location abstraction of entry and return states, PARTICLE instead adopts a strategy that correctly abstracts the vast majority of recursive calls but currently fails to handle a restricted class of legal programs. In particular, during the abstraction of function entry and return states, PARTICLE maintains a constant-sized simulated stack for each successive recursive invocation. For example, during the abstraction of function entry states, the symbolic evaluator places each successively general function invocation at the point on the stack at which the original activation record was located. This strategy permits straightforward abstraction of the function parameters and global state upon function entry, and allows the system to reach a fixed point. Unfortunately, the very characteristic that permits the process to reach a fixed point (the maintainance of a constant-sized stack rather than simulating the sequence of activation records built up during recursive function calls) also prevents the system from correctly handling programs in which the behavior of recursive function depends on values within the activation records of previous invocations of that function. (Examples of this phenomenon in C typically involve functions which pass pointers to their local variables to recursive calls to themselves.) Because the activation records associated with previous invocations of the recursive function are repeatedly overwritten in the process of abstraction recursive function calls, the state of this "downstack" data will not be correctly simulated or abstracted. The abstraction mechanisms used for generalizing a function's return state falls prey to similar difficulties. As a result, while PARTICLE's current methods for simulating the effects of arbitrarily many recursive function calls within a finite time work for the substantial majority of recursive functions, the current mechanisms are not completely general in their application.

Althought the method for abstracting recursive function invocations is currently incomplete, a slight modification of this method will yield a method that is fully safe and general technique. The fundamental difficulty with the current system stems from the fact that while a given location on the symbolic evaluation stack is being used to model the contents of arbitrarily many locations at different depths on the run time stack, at any given point the contents of that location only represents a legal approximation to the contents of a single one of those run time locations (namely, the deepest corresponding location currently on the stack). During the process of abstraction, the current method the contents of later locations simply *overwrite* the contents of earlier locations aliased to the same simulated stack slot. This can result in incorrect program behavior for cases where parameters or locals of earlier function invocations are referenced during the execution of later invocations of that function. A completely safe variant of this method would allow a given stack location to represent arbitrarily many run time locations, but would function only in such a manner that at any given point the contents of such a stack location would simultaneously represent a valid approximation to the contents of *all* run time locations mapped to that location. In such a framework, the data associated with later invocations of a function would not *overwrite* the data associated with earlier invocations of that function, but instead would coexist with that earlier data. Because the contents within a given simulated stack location is a legal approximation to the contents of *all* of the run time locations associated with that stack location, program behavior will always be simulated in a conservative manner. Moreover, because the representation of the data associated with arbitrarily many function iterations is captured within a finite amount of space and because the value domain is of a finite height, it is guaranteed that a fixed point of the abstraction process will eventually be reached.

The most straightforward implementation of the improved method would be safe but very imprecise: Given a technique that simply "folded" an arbitrarily deep call stack into a finite space, a given location in the innermost activation record would be aliased to many other locations in earlier activation records; the write of a known value to that location would result in the replacement of the current location contents with the LUB of the earlier location contents and the new value being written. As a result, even if a known value is written to a program quantity stored on the stack (such as a local variable or parameter), a subsequent read of that quantity following the write is unlikely yield back that same known value: Instead, the system will read a value that represents an approximation to the value read and the values stored in corresponding locations for the entire simulated call stack. Although it guarantees correctness and the termination of abstraction within some finite time, such a policy seems likely to very negatively impact the quality of program analysis performed within PARTICLE during the abstraction of recursive function calls and should probably be avoided even at the cost of introducing additional mechanism. Less naive strategies could capture the same important benefits while preserving the important capacity to perform strong program analysis for the innermost invocation of a procedure. One such strategy would adhere to the fundamental idea of "folding" the stack onto itself, but would maintain a special temporary area of the stack for the particular purpose of preserving information about the innermost invocation of the function. This temporary area would store *only* the data associated with that innermost invocation and could therefore be used to capture information about the values associated with the innermost stack frame without the need to take the LUB with the contents of other aliased locations upon every write. In order to maintain correct behavior and finite space usage for the stack representation, the buffer would be flushed ato the stack area and be reused upon recursive entry to another invocation of the associated. Although its implementation requires additional bookkeeping and complexity within the structure of the memory models associated with the symbolic evaluator, this technique would permit capture of much more precise about program behavior while guaranteeing that a fixed point of the abstraction process will eventually be reached.

4.3.6 Conclusion

The sections above have sketched the basic manner in which symbolic evaluation is conducted and has presented the fundamentals of the algorithms used during this process. The purpose of symbolic evaluation is to collect information on possible program specializations; the next section turns to examine the way in which this information is used in making specialization decisions.

4.4 Algorithms: Specialization Decisions

Section 4.2.4.1.2 sketched the fundamental basis for performing the specialization of a function using information collected about the sets of legal specializations associated with different calls to that function. We will briefly review these issues here, and will then turn to consider an algorithm to make high quality specialization and re-use decisions.

4.4.1 Overview

Conceptually, specialization decisions for a single function represent a *partitioning* problem: We are given a set of calls to the function[28], and wish to partition these calls into equivalence classes, where each equivalence class corresponds to a distinct specialized version of the current function and where the presence of a given call within an equivalence class indicates that the target of that call is the specialized function associated with that equivalence class. This view of specialization is illustrated in figure 4-15.

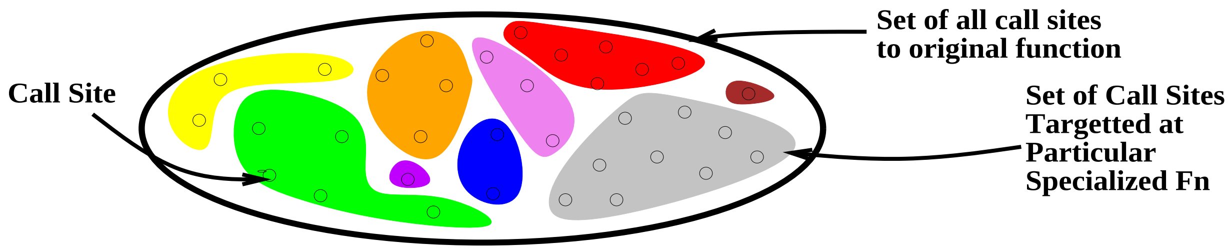
Call Site
Set of all call sites to original function
Set of Call Sites Targetted at Particular Specialized Fn

Figure 4-15: This figure depicts function specialization as a partitioning of the call sites to some original function into equivalence classes of call sites which allow similar sets of possible specializations.

4.4.2 Retargeting Call Sites

Pending the creation of any mechanisms for the "dynamic revectoring" of calls that would allow the same call site in a program to call several different functions at different times (see section 5.6.3.2.3.), we must treat the individual *call site* (rather than the single *call*) as the atomic unit that we can allocate to a particular specialized function. That is, when we wish to specialize a function and try to balance the benefits and costs of creating *specific* specialized functions, while we have information about the possible specializations and their respective savings for *particular calls*, we cannot simply associate individual calls from the same call site with different specialized function. Instead, *all calls from a single call site in the specialized function* must be targeted to a single specialized function.[29]

[28]It should be stressed that each of the calls being partitioned is *a simulated call during symbolic evaluation* that may itself represent many run time calls (e.g., if the call is located within a loop that cannot be simulated on an iteration-by-iteration basis).

[29]It is important to stress that because of specialization, a single call site in the original program may lead to *several* corresponding call sites in the specialized program. While it is true that all calls from a single call site in the specialized program must be targeted at the same specialized function, it is not true that all calls from a single call site in the original program have have the same target function.

4.4.3 Approximating Cross-Function Dependencies

The fundamental advantage offered by dynamic revectoring of calls lies in the ability of the calling code to take advantages of specialization of the caller (to allow different calls from the same call site to call different specialized versions of the callee). Given the lack of ability to perform a dynamic revectoring of calls, it would be ideal if PARTICLE had some ability to propagate information about specialization options and pressures between functions *during the specialization process*, so that the cost/benefit estimates associated with the specialization decisions for a certain function would be influenced by and would influence the cost/benefit judgments for other functions. Such a mechanism could serve as an alternative manner of obtaining some of the benefits of dynamic revectoring of calls without the associated complexity. For example, consider a case in which there are two distinct sets of calls to function B *from the same call site in function* A that each allow very different but lucrative patterns of specialization for function B. It would be ideal if we could create specialized versions of function B for each of these different sets of calls. (See figure 4-16.) Without revectoring, we can only retarget calls on the granularity of the call site. As a result, if we wish to create different specialized versions of B corresponding to each of these coherent patterns of calls from A, we must somehow persuade A itself to split in such a manner that the calls to different specialized versions of B are placed from placed from distinct calls sites in *different specialized versions of* A. By propagating information on cost/benefit judgments between different functions in cases such as this, it might be possible to apply some pressure on a caller procedure to specialize in a manner that would be advantageous to the specialization of the callee. Similarly, if a caller function A seems likely to be specialized along certain lines, the functions *called* from A can benefit from the knowledge of this information (by giving extra weight to partitioning schemes that take advantage of this information.) In both of these cases, the communication of cost/benefit judgments for a given function to its neighbors in the call graph could allow higher-quality optimizations than might otherwise be possible.

PARTICLE does not currently support the system of cross-function dependencies sketched above. While this remains an area whose full exploration is still possible (see section 5.7.2.), it seems highly questionable whether the significant (and perhaps enormous) compile time computational overhead associated added by the cross-coupling between function specialization decisions will be compensated by a significant increase in run time performance. In particular, explicitly modeling the cross-coupling means that a small perturbation in a specialization decision can potentially have ramifications for many different functions. At the least, the partitioning algorithm must check for the possibility of cascaded effects due to small perturbations. As will be seen below, the computational overhead associated with the separate optimization of the specialization decisions for each function in turn (without accounting for cross-function coupling of specialization decisions) can be very substantial. Although no experiments have been performed, it may well be that the optimization of a system which attempts to take into account the cross-coupling between specialization decisions in a system of any size would simply not be feasible (or because of the *very* high dimensionality of the parameter space, the heuristic optimization could conceive settle on less effective solutions than would be obtained by simplistic decomposition of the problem into independent optimization problems).

PARTICLE hopes to capture all of the intra-functional benefits and some of the inter-functional benefits of a more tightly coupled partitioning method without suffering the computational overhead associated with explicit modeling of the coupling between function special-

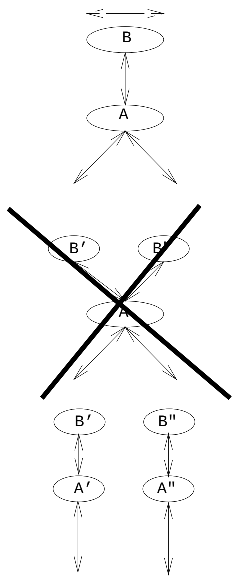

Figure 4-16: This figure illustrates the manner in which the specialization decisions for one function can influence the decisions for other functions. Due to different patterns of usage and possible specializations associated with different sets of calls from the *same call site* in function A, it is desirable to split function B into two subpieces. Unfortunately, it is not currently possible to dynamically revector calls from a given call site to different targets. In order to take advantage of the performance benefits associated with the specialization of function B, one thus has to split function A into corresponding specialized functions. Thus pressures promoting the specialization of B should also put pressure on the system to specialize A in the appropriate manner.

ization decisions. The approached adopted by PARTICLE is described below.

4.4.4 The PARTICLE Specialization Algorithm

4.4.4.1 Overview

Aside from the necessity of adhering to the overall constraints on legal specialization outlined in earlier sections, the structure of the algorithm used by PARTICLE to make specialization decisions is dictated largely by practical concerns. As noted above, rather than using a partitioning algorithm in which there is a great degree of complicated coupling between specialization decisions for different functions, PARTICLE specializes each function in isolation. Although this prevents us from taking *explicit account* of the possible benefits for another function of certain patterns of specialization in the function currently being specialized, by ordering our specialization decisions among functions in an appropriate manner we can still allow the specializations decisions for a given function to take advantage of (although not influence) previously made specialization decisions for another function. As discussed in section 4.4.2, the lack of any mechanism to perform dynamic revectoring of function calls constrains PARTICLE to specialize functions on the basis of patterns of usage for an entire call site. By making the specialization decisions for functions in such a manner that a given function is typically processed before the functions that it calls, we can substantially expand the set of *call sites* for the called function (since calls once originating from a single call site in a caller function are now spread among many call sites in different specializations of this caller function). Because of the spreading of calls to this function that were originally made through a single call site among many call sites, the groups of calls from a given call site in the specialized function now tend to be smaller than the calls associated with the corresponding call site in the original code. Moreover, each specialized call site in the caller function is found in some specialized function, which presumably is therefore itself associated with a certain pattern of usage and somewhat similar execution contexts (because specialized functions exist precisely to take advantage of such patterning). In light of this fact, it seems rather plausible that the calls from a given specialized call site to the current (yet unspecialized) function share certain patterns of usage that tend not be common to all such specialized call sites. Thus, the imposition of a partial ordering on the process of function specialization such that a given function will be specialized before the functions it calls can lead to a "spreading out" of sets of calls to the current function among many specialized call sites, thereby decreasing the number of different calls whose characteristics must be accommodated within the specializations performed for given call site *and* to the grouping of calls into call sites according to similar patterns of usage and behavior. As a result, this ordering can significantly relax the specialization constraints for the current function and may permit PARTICLE to take advantage of more fine-grained usage patterns than would otherwise be possible.

4.4.4.1.1 Heuristic Partitioning Because of the computational complexity inherent in any partitioning problem, it is not in general feasible to attempt to find *the* optimal solution for a reasonably sized instance of the problem. Instead, we make use of a general-purpose heuristic algorithm commonly used in combinatorial optimization: Simulated annealing. This algorithm yields tractable performance for the partitioning problem without the sensitivity to local minima that plague many simpler algorithms.

4.4.4.1.2 Coupling Between Specialization Decisions Just as PARTICLE avoids the overhead that arises from explicitly modeling the logical coupling between function-level specialization decisions, PARTICLE also attempts to simplify and accelerate the specialization process by ignoring the coupling between statement- and expression- level specialization decisions. In particular, while the cost metrics used to decide whether to perform or ignore a given specialization for a specific function should in theory take into account the likely presence or absence of other specializations in that same function (which might make the particular specialization being considered more or less desirable, or even unneeded), for the sake of computational efficiency PARTICLE models such specialization decisions as orthogonal and associates each specialization with a fixed cost/benefit judgment.

Given that the cost/benefit information associated with a specialization cannot be parameterized by information concerning the use of other specializations, PARTICLE attempts to be somewhat intelligent about the choice of this information. In particular, a common difficulty in the current system is that it models the benefits of different specializations as *additive* although the adoption of some of those specializations may serve to greatly decrease the benefits of others (and perhaps make those other specializations entirely useless).

For example, if an expression is capable of being evaluated entirely at compile time and being constant folded, it is likely that many of the subexpressions of this expression have also been constant folded through the application of separate specializations.[30] While it may seem natural to credit each constant folding specialization with the elimination of the time and space costs associated with its entire associated subtree, because PARTICLE treats each specialization within the expression tree as as orthogonal such an estimate represents an upper bound on the benefits of that specialization. In particular, there tends to be a high correlation between the applicability of specializations enabled at a given node and those enabled in subtrees rooted at that node and when considering which optimizations to perform, frequently the specialization algorithm will add together savings estimates that should *not* be treated as additive (because one such specialization eliminates the need for the others). For instance, in the example given above the algorithm selecting the set of optimizations to implement for different specialized functions will likely select *all* such optimizations as a unit, and will mistakenly believe that it has thereby eliminated considerably more time and space costs than is actually the case because the cost savings resulting from the elimination of a given subexpression will be counted several times (once for each enclosing expression that was constant folded). As suggested by this example, the naive use of this higher bound as the expected benefit for a specialization is rather unrealistic given the limitations of PARTICLE's mechanism for selecting specializations; such a savings estimate is particularly likely to yield poor results in cases in which a number of non-orthogonal specializations are capable of being applied simultaneously, and where the benefits perceived as resulting from such simultaneously application of specializations will be far greater than the actual benefits thereby accrued.

A lower bound on the savings associated with a specialization can be obtained by computing the cost of the tree being eliminated with the assumption that whenever the current specialization can take place, all specializations proposed within the subtree rooted at that node *will also take place*. This assumption provides a lower bounds estimate on the actual benefit of a specialization since it assumes that in all cases in which the specialization is applied *all* of the other possible "subsidiary" specializations that decrease the benefit of this specialization will also be applied.

This estimate will tend to downplay the benefits accorded by the specialization, and is unrealistic in cases where it is *unlikely* that the subsidiary specializations will be chosen for simultaneous implementation with the current specialization. However, a bit of reflection reveals that while this estimate indeed represents a lower bound on the benefits of a specialization, it is less unrealistic than might be thought. In particular, if the system makes use of information about specialization compatibility from several different instances of this specialization, it is possible to gain some more realistic feel for the actual benefits likely to be result from a given use of this specialization. To understand why this is, it should be pointed out that there are two primary reason that a group of specializations discovered together during processing would not all be chosen for simultaneous implementation within a specialized function: Some of the specializations may be judged as too costly, or some of those specializations may not

[30]This is not strictly true, for the constant folding some expressions does not require exactly known values – e.g., $(a + b) > 0$ may be constant folded even in a range of cases where the precise values of a and b are not known.

be legal in the contexts for which the specialized function is being prepared. Although the first reason for not exploiting a possible specialization can be important in cases in which space is tightly rationed[31], it applies only to one of the specializations currently performed by PARTICLE (loop unrolling), and is ignored within the heuristics adopted by PARTICLE to assign cost/benefit information. The second condition is a far more common and fundamental reason for the inability to simultaneously perform two distinct specializations; fortunately, the presence of such a condition is detectable during symbolic evaluation and the symbolic evaluator can adjust its estimates of the benefits associated with a given specialization accordingly. In short, by recognizing cases which suggest that the subsidiary specializations may not always be legal when the primary specialization is used, the symbolic evaluator can upwardly adjust its estimates of the benefits likely to result from the primary specialization. The idea behind this adjustment process is simple: If the subsidiary specializations are not capable of being used in all conditions in which the primary specialization is used, there must be conditions during symbolic evaluation in which the primary specialization is discovered to be applicable, but the subsidiary specializations are not possible. In these cases the lower bound on the benefit of a specialization that was discussed above will tend to be be higher, reflecting the fact that the specialization is likely to be more valuable.[32] By taking advantage of the assumption that subsidiary specializations will tend to be used in conjunction with a subsuming optimization wherever possible, and the fact that a minimal precondition for the consideration of the use of an optimization in a specialized function is its applicability within *all* execution contexts resulting from a given call, we can create a more realistic approximation to the benefit associated with a specialization. In particular, we can take the *maximum*[33]. of the lower bound benefit estimates estimating the amount of savings possible for that specialization within all execution contexts *associated with a given call* as some approximation to the actual benefit that would result from performing that specialization.[34] Because function specialization typically takes place in such manner that several calls are associated with a given specialized function, better estimates yet can be obtained by averaging together the benefit estimates associated with a particular specialization that is possible within the context of several calls. Although such the maintainance of such estimates requires some amount of machinery beyond that needed for purely local cost estimates, the additional overhead is rather modest.

Although the methodology presented above helps to address some of the inaccuracies associated with the assumption that specialization benefits add in cases where one specialization subsumes another, there are other cases not addressed by the above methods where the as-

[31]In the current version of PARTICLE, the set of specializations used is such that the only possible *costs* of a specialization are due to increased space usage.

[32]The estimated benefit will increase because some subsidiary optimizations are not possible, and assuming that all such optimizations are possible will therefore have less of an impact on the benefit estimate for the primary specialization.

[33]The fundamental idea here is that a subsidiary specialization must be applicable in *all* execution contexts for a given call to be considered for use in a specialized function; if a given subsidiary specialization decreases the benefit of the specialization being considered in one case but not in another, that specialization must not be applicable in all contexts, and its constriction of the benefit estimates for the specialization being considered can thus be ignored – hence the use of the *maximum*

[34]In essence, such a procedure allows us the symbolic evaluator to “low-pass filter” the lower bound benefit estimates associated with a given specialization, and to avoid underestimating the benefits associated with that specialization simply on account of transient specializations that cannot actually be used.

sumptions of specialization orthogonality used by PARTICLE lead to unrealistic cost/benefit judgments. For example, consider the proposed specialization of a conditional statement that collapses one of its branches of that statement. The space savings attributed to that specialization reflects the space that would be saved by the elimination of the original code for that branch. In fact, in the absence of this specialization, it may be that the branch being eliminated would offer considerable opportunities for specialization so that the space savings estimate associated with the specialization is rather inaccurate.

4.4.4.2 Cost/Benefit Information

While making specialization decisions, PARTICLE explicitly weighs the costs and benefits associated with different possible patterns of specialization. The information used in this process is collected during symbolic execution, and may be either exact or estimated where insufficient information exists to allow exact statistics to be deduced. The per-function information used in this process is described below; the mechanisms by which this information is combined in order to make decisions regarding specializations are discussed in the next section.

- The set recording all calls to this function.
- For each call to this function, information regarding the
  - Set of all specializations that are applicable for this call, and their
    - Estimated Time Savings/Cost (for this particular call). This savings/cost is the estimate for the total savings/cost due to this particular specialization for a specific call to this function. (e.g., If the specialization is in a loop, the savings for a particular execution of the specialization is multiplied by the appropriate estimated or empirically determined factor.)
    - Estimated Space Savings/Cost (for this particular call).
  - The set of statements executed for this call.
  - The call site from which this call originated.
  - An estimate for the number of run time invocations represented by this call. (e.g., If this call was performed in the process of abstracting a loop, this call may be taken as representing the calls for *all* iterations of that loop.)
  - The identity of the "parent call" to this call (i.e., the call associated with the invocation of the parent function that in turn invoked this call). Because the original caller function may itself be subdivided into many specialized functions (each associated with its own set of calls to *it*), this allows the determination of which of those specialized version of the caller will originate the call associated with this entry.

Note that although PARTICLE revectors function calls on the granularity of *call sites*, we cannot simply condense the per-call information above into per-call-site information during data collection: The call sites available during *function specialization* will in general be a superset of those available during symbolic evaluation (simply because, as discussed above, some of the call sites for functions that called the function currently being processed may have themselves undergone mitosis via incorporation into different specialized functions).

4.4.4.3 Specialization Algorithm

4.4.4.3.1 Ordering of Function Specialization The specialization algorithm used by PARTICLE begins by creating a directed call graph for the input program. Each vertex v in this graph is associated with some function in the original program. An edge extends from vertex i to vertex j iff there exists a call in the user program from the function represented by vertex i to the function represented by vertex j. We would like to impose an ordering on the functions of the program, such that if function f calls function g, then we specialize function f prior to the specialization of function g. (As described above, this should lessen the constraints on specialization of g and permit a more finely specialized version of g by "spreading out" the calls originally associated with a given call site in f among many different call sites located in specialized versions of f.) Unfortunately, in the presence of recursion this graph can contain cycles and no such ordering of functions exists. (e.g., There can be functions f and g such that f calls g and g calls f.) In the presence of subgraphs with such cycles, PARTICLE chooses to simply "break the cycle" at a natural point and impose a convenient ordering. Thus, the general strategy is as follows: We construct a DAG from the call graph by performing a depth-first search of the graph and omitting back-edges. Performing a topological sort on this DAG provides us with the partial ordering among vertices (and thus functions) that we desired. Given the ordering among functions, we simply proceed to specialize each function in turn.

4.4.4.3.2 Specialization Decisions for a Single Function As noted above, the specialization decisions for a single function can be cast in terms of a *partitioning* problem: We would like to partition the call sites of the original function up into equivalence classes of revectored call sites. Each such equivalence class corresponds to a set of call sites which makes use of a single specialized version of the original function (presumably specialized so as to take advantage of particular patterns of Values associated with calls from these call sites). It is worth stepping back to review the costs and benefits associated with the creation of a specialized function.

4.4.4.3.2.1 Benefits and Costs of Specialization The fundamental benefit of the creation of a specialized function is the reduction in run time cost offered by many specializations. As part of their character as "reductive" optimizations, most specializations serve to shift computation away from run time to compile time, and sometimes the cascaded savings can be substantial (e.g., the reduction of a call to a sophisticated function call to a constant). Further (perhaps second-order) benefits may arise from the enlarged size of basic blocks produced by the elimination of conditionals, and improved cache performance due to the use of more compact code.

Just as the benefits accruing from specialized functions are found mostly in the performance (time) domain, the *costs* associated with performing such specializations are generally lie in the increased space demanded by such functions. In particular, it is often the case that the space necessary to represent a single function associated with a certain set of call sites is less than the space necessary to represent two (or more) functions each specialized for use by particular subsets of those call sites. Note while the above situation frequently obtains, it should be stressed that there are certainly some cases in which it does *not* – In particular,there do exist cases in which the splitting of a specialized function into two distinct specialized functions can *save* significant space: As a trivial (and somewhat unrealistic) example, consider the possibility of replacing a version of a square root routine that is specialized for two possible arguments with two square root routines specialized for particular known arguments, as illustrated in figure 4-17. Each such specialized routine will require minimal space to represent – far less than half of the code that is needed to represent the single, more general function. Nonetheless, in practice such cases are likely to prove the exception rather than the rule – particularly since this requires that the *sum* of the space required for the representations of all of the new specializations be less than the space required by a single specialized function that would be associated with *all* of the calls.

Thus, while the benefits of creating specialized functions lie in improved run time efficiency, there can be substantial space costs associated with the proliferation of such functions. In keeping with its philosophy of providing the user with a wide array of controls over the quality, character, and speed of the specialization process, PARTICLE allows the user to specify their subjective judgement of the ratio of the undesireability of a unit estimated space cost to the subjective desireability of a unit estimated time benefit. This preference is then used as an explicit parameter in the specialization algorithm to inhibit or encourage the creation of specialized functions.

4.4.4.3.2.2 Overview of PARTICLE's Partitioning Strategy The algorithm used by PARTICLE to specialize a particular function works in two stages. Both of these stages involve attacking combinatorial problems using the simulated annealing method [32].

The first stage of PARTICLE involves the creation of a tree of call sites organized such that call sites supporting similar sets of specializations are located nearby in the tree. This tree (a "clustree") is used to vastly constrain the set of possible partitionings whose time and space costs must be explicitly weighed during the second stage.

The second stage of PARTICLE subdivides the tree created by the first stage into a set of subtrees, each of which represents a particular specialized function. While stage 1 constructed the clustree only on the basis of the similarity of sets of specializations permitted by different call sites, stage 2 explicitly considers the time and space costs of creating a specialized function for a particular set of subtrees, and attempts to arrive at a low-cost and high-benefit partitioning.

It should be noted that the subdivision into these two subcomponents was in part motivated by a similar subdivision used to implement clustering analysis in [52]. While the second stage used in [52] makes use of Minimum Description Length theory to find a level clustering for the clustrees output by the first stage, PARTICLE must take into account the time and space tradeoffs associated with different partitioning schemes, and thus cannot make use of MDL.

4.4.4.3.2.3 The Simulated Annealing Framework The technique of *simulated annealing* [32][43] is a physically motivated method for finding approximate solutions to combinatorial problems whose size and character precludes direct examination of all possible solutions. Simulated annealing is particularly designed to allow computational tractability while avoiding the tendency of purely locally-oriented "gradient-descent"-type greedy algorithms to reach merely local cost minima. In particular, the system follows a "mostly downhill" approach to minimize the cost (or "energy") function, but is associated with a significant stochastic ("temperature") component that permits occasional transitions from a low-energy to higher-energy state. This temperature is slowly decreased over time, so that more global exploration is encouraged at earlier stages while later stages allow a closer approximation to gradient-descent

```
/*  routine specialized for either of two different positive arguments */


        double sqrt(double  d)
        {
            {
              double dNew;
              double dOld;
              double dAbs;

              dNew = d;
              dOld = 0.0L;

              dAbs = abs(dNew - dOld);

              while (abs(dNew - dOld) > SQRT_EPSILON)
                {
                  dOld = dNew;
                  dNew = ((d / dNew) + dNew) / 2.0L;
                }

              return(dNew);
            }
        }


        /*  individual routines each specialized for particular positive argument
            after dead code removal
         */


        double Specializedsqrt0(double  d)
        {
          return(10.0);
        }

        double Specializedsqrt1(double  d)
        {
          return(3.0);
        }
```

Figure 4-17: This figure presents a case in which the partitioning of a single function into several specialized functions can actually save space. In this case, few specializations are legal for *both* of the two particular uses of the routinw in the program. On the other hand, if the function is specialized with respect to each of these arguments independently, enormous optimizations can be performed.

type methods. Such methods have a physical analog in the cooling of liquids into solids: While slowly cooling a liquid allows atoms time to explore the parameter space and to associate into a large-scale minimum-energy crystalline configuration with remarkable reproducibility, "quenched" liquids are associated with higher-energy "amorphous" or "polycrystalline" final states that are merely locally optimal [43].

The application of simulated annealing to a combinatorial system requires

- A state space for that system (capturing the range of different parameters to be optimized).
- The existence of a cost or energy function computable for every point in the system state space.
- A position in the system that represents our "current state" in the exploration of the state space.
- A process to generate random perturbations to the current position in the system.
- An *annealing schedule* that provides for a slow "cooling" of the system over time.

In the following few sections, we will turn to see how PARTICLE makes use of this framework in optimizing the partitioning of call sites into groups associated particular specialized functions.

4.4.4.3.2.4 Clustering of Call Sites The first stage in the specialization algorithm groups together similar call sites in the framework of a hierarchical "clustree". The clustree is isomorphic to the "dendogram" that has long been applied in traditional hierarchical clustering algorithms [52], and which represents the hierarchical clustering of a set of data points. All data stored by the clustree is located on the leaves; and interior nodes n represents the set of all data points within the subtree rooted at n. Within PARTICLE, the leaves of a clustree represent individual call sites, and the clustree itself describes a hierarchical association of these call sites. By organizing the call site data in this manner, the partitioning decisions associated with the second stage of the specialization algorithm become much simpler. It is important to stress that the clustree produced by the first stage of the algorithm does not impose any particular partitioning on the function, but merely serves to *constrain* the range of possible partitionings to be considered by the second stage.

The creation of a minimal-energy clustree (one in which the most similar data points are nearest one another) is computational intractable (and is a close relative of variations of the NP-hard parsimony analysis problem), finding an optimal solution is thus in general not feasible. Instead, PARTICLE makes use of the simulated annealing framework method described above. The parameters in the optimization process are as follows:

- The state space associated with the specialization procedure is the set of all possible binary clustree whose leaves represent the call sites for the given function.
- The cost function which we wish to minimize is the gaussian entropy of the clustree. The motivation for using an energy function based upon this metric arises from analogy with clustering problems, and particularly from the Numerical Iterative Hierarchical Clustering

framework presented in [52]. Intuitively, minimizing the gaussian entropy yields a tree exhibiting minimal variance (where the notion of variance is extended from a single data point to an entire tree structure as in [52]). Intuitively, subtrees in which call sites associated with similar patterns of possible specializations are located nearby will tend to have low variance, whereas the variance for a subtree with very disparate call sites in the leaves will tend to be very high.

- The "current state" in the optimization system is simply the currently proposed clustree, whose leaf nodes correspond to the call sites for the given function.

- During the optimization of the clustree, random perturbation to the current state are implemented in the form of "grab" [52] operations performed on the clustree. The "grab" operation on clustrees was first applied in the context of many-body simulations [3], and has been used more recently in the context of Numerical Iterative Hierarchical Clustering [52]. Conceptually, the grab operation shifts a subtree of nodes from one portion of the clustree tree to another while conserving the number of nodes in the tree and the identity of all leaf nodes. In particular, *grab(d,s)* creates a new binary tree in which nodes s and d are siblings in a new subtree placed at the original location of node d, and where the old parent of s is replaced by the sibling to s. In a clustree, each interior node n represents some set $S(n)$, consisting of all call sites (represented by leaf nodes) in the tree which are descendants of node n. Following a grab operation, all for all nodes x along the path from the grandparent of s to the first common ancestor of s and d, $S(x_{after}) = S(x_{before}) - S(s)$. Similarly, for all nodes y along the path from the grandparent of d to the first common ancestor of s and d, $S(y_{after}) = S(y_{before}) \bigcup S(s)$. This perturbation mechanism permits us to rearrange the contents of the clustree at different scales. At each point in the annealing, we simply pick a random pair of nodes on which to perform the grab operation, subject to the constraint that one node cannot be the ancestor of the other.

- We make use of an empirically tested and prepackaged annealing schedule commonly used in combinatorial optimization. The user may adjust the schedule to permit for faster or slower cooling.

It should be noted that past projects have applied simpler, greedy methods to good effect in the context of clustree optimization, and it remains to be seen whether the use of simulated annealing yields sufficiently better results to justify its additional computational cost [52].

4.4.4.3.2.5 Partitioning of the Clustree The first stage of the function specialization process creates a clustree representing the calling sites for the current function in which nodes representing calling sites that are associated with similar patterns of possible specializations are located nearby one another. The second phase of the specialization process accepts as input the clustree output by the first part, and partitions that clustree in such a way as to maximize the overall benefit of specialization for the current function. In particular, the partitioning algorithm heuristically explores the space of possible partitions, noting for a particular proposed partitioning the overall time savings and space cost associated with that partitioning. The user-defined metric described above is used to judge the "goodness" of that partitioning on the basis of this space and time savings and cost information. The reverse of this "goodness" metric is used to define the cost function we are attempting to minimize. As was the case for the optimization of the clustree, finding solutions to the clustree partitioning problem that are known to be optimal is not computationally feasible, and PARTICLE makes use of a simulated annealing method to find high quality solutions to the problem within an acceptable period of time.

The parameters to the simulated annealing framework used to optimize the partitioning for specialization are described below:

- The state space for the optimization is the set of all possible partitions of the clustree. Since the leaves of the clustree correspond to individual call sites, each partitioning of the clustree implies a partitioning of the call sites. (See figure 4-18.) As noted earlier, each such partitioning corresponds to a set of call sites vectored to a particular specialized version of the current function.

- The cost function for the system is the cost associated with a particular partitioning, as calculated from the information collected during program analysis. The effect of a perturbation on this cost function can be efficiently incrementally computed; we need not completely recalculate the cost for each point in state space when performing the optimization.

- Our current state in the system during the process of optimization is simply the currently proposed partitioning.

- The random perturbations we perform to explore the state space map one partitioning of the clustree to another such partitioning, and perform one of two functions: Either splitting one partition into two pieces (by removing a subtree from that partition) or merging together two distinct partitions into a single partition. As discussed above, each clustree partitioning corresponds to a partitioning of the call sites into sets, each of which is targeted at particular specialized function. Thus, the perturbations correspond to the replacement of a single specialized function by two separate functions each used by some disjoint subset of the call sites for the original function, and to the replacement of two distinct specialized functions by a single function serving all of the call sites originally associated with *either* of the two original specialized functions. In order to gain a feel for the optimization process, it is worth pausing to consider the effects of each of these actions: If PARTICLE decides to replace a single specialized function by two distinct functions, this might allow a significant run time savings, since the two new functions may be able to take advantage of particular specializations that are legal for the particular subsets of call sites associated with each of these new functions, but not legal for *all* of the call sites of the original function. On the other hand, in many cases the creation of two such specialized functions rather than a single function will have significant space overhead, although (as noted in section 4.4.4.3.2.1.) this is not always the case. Conversely, while merging together the call sites associated with two specialized functions may permit substantial space savings, it may also lead to degradation in the quality of optimizations allowed for the function being specialized. All of these costs and benefits (or approximations to them) are weighed by PARTICLE when judging a particular partitioning scheme.

- As in the application of simulated annealing for the calculation of an appropriate clustree, PARTICLE makes use of an empirically tested but tunable annealing schedule in the partitioning algorithm.

The partitioning algorithm associated with stage 2 of the function specialization process thus works by heuristically exploring the set of possible partitions of the clustree output by stage 1 (corresponding intuitively to a set of "reasonable" partitions targeting different sets of call sites to distinct specialized functions). The cost/benefit calculations make use of an explicit, user-defined metric of "goodness"; by creating metrics that give different relative weights to space costs and time savings, the user can change the emphasis of specialization, preventing the introduction of specialized functions in all cases not likely to be associated with tremendous performance improvement, or allowing the creation of specialized functions to aggressively exploit the opportunity for even a slight performance gain, regardless of the space constraints.

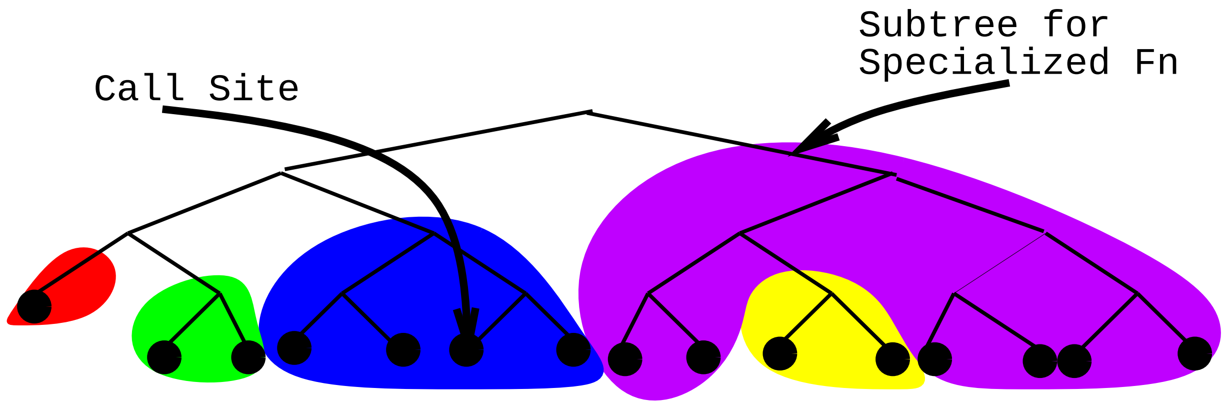

Figure 4-18: The Clustree output by the first stage of specialization is arranged so that call sites allowing similar patterns of specialization are located nearby in the tree. The second stage of the specialization procedure partitions this tree of call sites into subtrees corresponding to groups of call sites each associated with a particular specialized function.

4.4.5 PARTICLE's Specialization Algorithms: Conclusions

Unlike previous on-line evaluators which made specialization decisions dynamically during symbolic evaluation, PARTICLE weighs options for specialization only once full information is available about program behavior. This section has reviewed the means in which PARTICLE uses this information to guide the specialization decisions. Although the size and character of the specialization process rules out the possibility of deriving solutions that are guaranteed to be optimal and prevents direct modeling of the dependencies between the specializations of different functions, PARTICLE makes use of a powerful heuristic optimization framework for the specialization of individual functions. This process avoids the local minima associated with greedy strategies and directly weighs specialization decisions in light of user-specified space cost/run time benefit preferences.

4.5 Conclusion

This chapter has sketched the basic mechanisms and algorithms underlying PARTICLE's operation. We examined PARTICLE's functioning during symbolic evaluation, its simulation of program expression using Values drawn from its abstract domain, and its general handling of control flow constructs in the presence of possible ignorance about actual patterns of run time control flow. The purpose of this high-fidelity simulation of program behavior is to allow maximal discovery of possible program specializations, and opportunities for possible program specialization are noted throughout the process of symbolic evaluation. The later sections of this chapter examined how the information on possible specializations that was collected during symbolic evaluation is used to create a specialized output program that during a process explicitly guided by careful consideration of cost/benefit tradeoffs whose character is set by the user.

Chapter 5

Future Extensions to PARTICLE

5.1 Introduction

Previous portions of the thesis have outlined the structure of the PARTICLE abstract interpretation framework and have sketched the design and operation of the system currently implemented in that framework. This section turns to consider a few of the diverse ways in which the current system could be improved. The suggestions below vary widely in emphasis: Some suggestions concentrate on improving the efficiency of the symbolic evaluator while others discuss varied means by which the quality of the program analysis can be improved. Certain of the suggestions discuss architectural changes that modify the fundamental character and operation of the analysis performed, while others limit themselves to proposing incremental improvements to the current system. In many cases, the suggestions are completely orthogonal in emphasis and implementation. This chapter is organized in a manner that reflects the logical character of the suggestions, and hierarchically subdivides the proposals according to their different emphases and motivations. PARTICLE is a large system, and many facets are open to changes. This chapter limits the discussion to those proposed changes whose expected rewards are judged to be the highest; because of space constraints even such suggestions can only be granted the most cursory introduction.

5.2 Lacunae in The Support of C and Crucial Library Routines

Although PARTICLE supports a very popular and powerful subset of the C language, there are some significant gaps in its language support. In some of these instances (e.g., support for multiple program source files, static variables, unions, and aggregate objects whose size is not known at compile time), such gaps in support reflect little more than the temporary neglect of small and rather unimportant implementation efforts until the completion of the basic system. In other cases, failure to support a feature of the C language (or its commonly used libraries) is based upon the perception that support for this feature would require a considerable effort in implementation or design. The omissions of the current PARTICLE system that pose the greatest design or conceptual challenges are discussed briefly below.

5.2.1 Handling of General Control Flow

The presence of *goto* statements within C permits the realization of arbitrary control flow graphs within a function. Although traditional iterative flow analysis algorithms have no difficulty in handling general flow graphs, the process of loop abstraction can be awkward and time-consuming in the presence of control flow of even moderate complexity (see section 4.3.4.). Although a program specializer based on symbolic evaluation can effectively operate on large bodies of C code without any need to handle such unusual patterns of control flow, it would be a substantial advance to derive a general purpose mechanism to handle such cases in a graceful manner. The use of such a technique could permit abstract interpretation to be smoothly and aggressively applied in other low-level languages and domains, and would constitute a substantial step towards a unification of abstract interpretation and standard flow analysis techniques.

5.2.2 Modeling Aggregates of Incompletely Known Size

A shortcoming of the current PARTICLE implementation not discussed within earlier sections of this document is the system's inability to handle of the allocation of arrays of aggregate data whose size is not precisely known during symbolic evaluation. While this lacuna is "conceptually superficial" in the sense that its resolution requires merely a careful set of relatively minor implementation changes rather than any large-scale conceptual changes in the structure of the system, the modifications to be made will require some consideration. In particular, the fundamental challenges associated with the handling of objects of unknown size lies in the need to consistently and efficiently model the object with maximal precision in the context of operations upon the memory model.[1] The maintainance of information concerning aggregate size is primarily useful for two reasons: For bounding the possible referents of pointers which pass over, in, or out of the aggregate, and for permitting more precise models to be constructed of the aggregate elements. Each of these motivations for size information is discussed briefly below.

5.2.2.1 Motivations for Aggregate Size Information: Maintaining More Accurate Models of Array Contents

Information regarding the size of an array can be used to permit the sharpening of models of the array contents. For example, if it is known that the size an array will be within some range, it may be desirable to create an array model which individually represents minimal set of array elements known to be found within the array (i.e., the set of array elements guaranteed to exist by the minimal array size), along with an additional model element approximating the values associated with all other possible array elements.[2] While this model of the array may

[1]Note that because of the limitations on the behavior of C programs, certain awkward situations do not arise: For example, there is never any need to generalize a memory model with a stack containing an object of unknown size but uniform contents against a stack of known elements. (An operation whose maximally precise handling could require substantial effort).

[2]Note that such a "model element" for the remainder of the array would likely be quickly LUBed to $\top$ within code of any reasonable complexity, so that the maintainance of such an approximation element for the remainder of the array is unlikely to offer much benefit for analysis.

be grossly incomplete, it may permit precise modeling of writes to the first sets of elements in the array and may be successively extended and enriched as increasing amounts of information concerning array size become available.

5.2.2.2 Motivations for Aggregate Size Information: Bounding the Results of Pointer Manipulation

The previous section briefly sketched the benefits offered to array representations by aggregate size information. This section turns to consider the employment of such information for sharpening knowledge about the possible referents of program pointers, but also considers conditions under by which semantic assumptions concerning the character of pointer manipulation can allow the deduction of information concerning the extent of segments (and, by implication, of arrays). In the absence of the fine-grained segmentation constraints described in 4.2.3.2.2, array size information may prove useful for bounding possible referents of pointers into the array or within nearby areas. For example, consider a pointer originally referring to some known point on the stack and subsequently decremented by some known amount that is known to carry it into (and possibly beyond) an object of incompletely known size. (See figure 5-1.). Although it would be possible to simply to declare the referent of the resulting pointer entirely unknown, a desirable alternative would permit the symbolic evaluator to bound the possible referents of the decremented pointer to some range of locations. Such additional information could help to minimize the amount of static information that must be invalidated upon an indirect write through that pointer. In cases where partial information is available about aggregate size and where the amount by which the pointer is decremented is statically known, such bounds can be relatively easy to obtain. In other cases (e.g., references where the size of the array is completely unknown), it can be considerably more difficult to establish the possible bounds on the referents associated with the decremented pointer. (e.g., In principle, it may be possible to symbolically relate the size of the movement to the size of the array being passed over, so as to prove that the pointer lands in the array, but such reasoning extends beyond the scope of the current PARTICLE system – see section 5.5.6.)

The presence of fine-grained segmentation constraints allows the assumption that any pointer manipulation on a preexisting pointer into an array will yield a pointer into the same array (and thus that any array reference at some known offset within the program represents a legal access to an element within that array). With the establishment of such assumptions, much of the motivation for externally bounding the results of pointer manipulation disappears: The bounding of pointer references is adequately performed by the application of segmentation rules. At the same time, the strength of such semantic assumption permits information concerning the size of allocated regions to be collected *from references* into such arrays. The information bounding array sizes can then be fruitfully used to create more accurate models of the array contents as discussed above. For example, when an element in an array is accessed at a known or partially known offset, the ability to make assumptions about segmentation permits the system to recognize that the array element being accessed must exit, and thus the index of that element provides a lower bound on the possible size of that array. Note, however, that like any other information concerning program values that is acquired after the original value was created, the *knowledge concerning the size of the array* must itself be generalized with the knowledge concerning that value from other contexts – even if the generalization is not strictly

```
{
        int w;
        int array[n];        /*  n unknown */
        int x;
        int *p;

        p  = &x;
        p -= 10;        /* where does p now point?  before, after,
                           or into the array?  */
        *p = 0;
}
```

Figure 5-1: When pointers are incremented or decremented in the presence of inexact knowledge about the size of program data objects, it may be difficult to determine the possible referents of that pointer.

necessary. (See section 5.5.3.) As a result, information regarding array size that is discovered on the basis of reference patterns may prove briefly useful, but ultimately ephemeral.

5.2.2.3 Conclusion

While the mechanics of modeling the program store in the presence of aggregates associated with incomplete size information are relative simple, high quality modeling of program behavior in the presence of such ignorance can be difficult and requires considerable additional mechanism and thought. In light of the care with which this task must be undertaken, it seems reasonable to with support for modeling aggregates of unknown size until the relevant issues have been thoroughly considered.

5.2.3 Approximating Operations for Heap Management

Most large C programs make intensive use of dynamically allocated storage. The mechanisms used for managing such storage are not part of the C language itself, but are available as part of the standard library routines that are a fundamental component of any C development environment. This section discusses the advantages offered by the maintainance of special-purpose mechanisms to model dynamically allocated heap structures and to simulate the behavior of heap-related library functions at a high level. It may seem somewhat odd to consider adding substantial mechanisms to PARTICLE to reason about a rather small set of library mechanisms when the symbolic evaluation system is already capable of full-bodied modeling of pointer behavior within C itself – it seems natural to wonder if the effects of the small collection of heap management routines cannot be directly simulated during symbolic evaluation.

Unfortunately, the low-level behavior of the heap management algorithms are highly data-driven. Because heap management is so data-driven, slight uncertainties concerning patterns of allocation and deallocation at any point during the execution of the program can lead to a situation in which future allocations will be regarded as potentially aliased and vast amounts of static heap data will be regarded as imprecisely known. For example, the central data structure of many heap management systems is a free list for memory blocks. The internal behavior of a given dynamic allocation routine will rely heavily on the data recorded within this structure, and the pointer returned by the allocation routine will not be precisely known unless the contents of the free list are known to a high degree of precision. Since the contents of the free list changes with every allocation and deallocation, it is likely that those contents will change substantially between different contexts within the program and the state of the structure is thus likely to be imprecisely known after program join points, resulting in an increasing cascade of subsequent allocations whose exact allocated regions are not fully known and whose free list is even less well known. Due to the lack of precise information concerning the location of free regions of memory, the simulation of the allocation algorithms will yield inexactly known return values. The inability to precisely specify the location of allocated regions will lead to the perception that the same area of heap memory is possibly associated with distinct allocations and will necessitate the modeling the contents of that shared area with contents that are legal approximations to the values written into the regions returned by each allocation. That is, a *particular* range of memory locations will be regarded as possibly associated with a set of different allocations, and will therefore be modeled by contents that represent a legal approximation to the region's contents under *any* of these possibilities. Similarly, because the referents of the pointers returned by heap allocation routines will typically be imprecisely known, few writes to heap structures will represent writes to known locations and few locations within the heap will contain known data.

The fundamental shortcomings of the low-level simulation of heap behavior arise from its attempt to describe the heap on a location-by-location basis. The fine granularity may succeed in expressing some information about the low-level location-by-location details of heap layout, it fails to capture the distinctions important to most program behavior. In particular, most programs manipulating heap structures tend to care not about the location-by-location layout and composition of the heap (e.g., which particular set of locations is selected for a given dynamic allocation, and which allocated blocks are immediately adjacent), but rely only upon the properties guaranteed by the specifications for the memory management abstractions. Such specifications provide no information on or guarantees regarding low-level heap layout aside from guaranteeing that all simultaneously allocated regions are contiguous, are of at least the requested size, and do not overlap. As a result, safe programs manipulating heap structures abstract away from the location-by-location composition of the heap and instead adhere to conventions that treat each dynamically allocated region as representing a unique set of locations independent of all locations associated with other past, present or future allocations, and regard deallocation of such a region as a destruction of the associated locations. Given the higher level at which heap management and use takes place, it becomes clear that location-by-location modeling of the heap is both needlessly expensive and unlikely to capture many important aspects of program behavior.

It may strike the reader as somewhat surprising to discover a domain in which high quality low-level modeling of program behavior is likely to actually *hurt* the quality of program analysis, and it is worth pausing to consider the underlying reasons for such difficulties. In general, the quality of program analysis suffers when there is a gap between the level at which programs operate and the level at which analysis takes place. Although it is clear why high-level program analysis will frequently have difficulty effectively modeling the effects of and optimizing low-level code, it may be less clear why low-level analysis will find it difficult to effectively simulate the operation high-level code. After all, it might be expected that while maintaining a wealth of data concerning details of low-level program state when compiling high-level code would consume unnecessarily large amounts of compiler resources, such rich low-level information would at least permit aggressive (albeit expensive) analysis and optimization of that code. Unfortunately, for low-level representations like those used in PARTICLE, this is not always true: In some situations, maintaining even precise low level information can fail to capture certain high-level invariants and characteristics of program behavior, while an appropriate higher-level decomposition of program state would both save significant compile time resources and permit the recognition and exploitation of those aspects of program behavior. It should be stressed that the inability of low-level information to sufficiently characterize some aspects high-level behavior stems not from any inherent shortcomings of low-level data, but from the failure of a particular class of low-level representations capture *all* of the low-level information relevant to such behavior. As discussed in section 3.2.2, even the detailed low-level information that is maintained by PARTICLE about program quantities can fail to capture very important aspects of program behavior. While it might naively be assumed that higher-level representations of static knowledge are "coarser-grained" than those offering low-level descriptions of such knowledge, this need not always be the case: In some cases, higher-level representations carve up the knowledge space in a decidedly different manner than low-level descriptions. As a result, information that may not be expressible by or captured in in low-level representations may be extracted from higher-level descriptions of static knowledge. For example, as discussed in section 3.2.2, PARTICLE's value representation little attempt to represent the logical links between the different values possibly associated with distinct program quantities. If it were feasible to maintain such information in addition to location-by-location data concerning heap contents, the quality of heap behavior analysis permitted could equal or exceed the quality afforded by higher-level analysis. It is conceivable that with a sufficiently general and powerful basis set of low-level information collected through program analysis, sufficient information would be maintained to void any need for any higher-level and coarser-grained program analysis. (In the terminology used above, it may be possible to formulate a form of low-level information that would carve up the space of static knowledge in a such manner that the low-level knowledge is truly a finer-grained version of the higher-level knowledge.) In certain circumstances, however, carefully chosen high-level information can efficiently and precisely capture high-level information concerning program behavior while offerer markedly improved analysis speed and vastly reduced compile time resource requirements.

Given the widespread adherence to disciplined patterns of heap usage and the difficulty of obtaining useful information concerning heap behavior from low-level program analysis, it is natural to consider the use of high-level data and semantic constraints concerning heap structure and use during symbolic evaluation. The ability to make use of high-level assumptions concerning dynamic allocation (such as the fact that an allocated region is guaranteed not to be aliased with any already existing region, and that all regions whose deallocation is requested were indeed previously allocated) permits the symbolic evaluator to avoid making grossly conservative assumptions about program behavior, and suggests a natural level at which to analyze program behavior. At a particular moment in time during the symbolic evaluation of a program, the heap can be represented as a series of disjoint allocated regions with no assumed connections between them. The series of locations *within* each such region can be represented with as much fidelity as locations anywhere else in memory – the representation simply avoids imposing assumptions about the existence or structure of a global space linking all allocated regions. Such a heap representation can be straightforwardly extended to allow for the generalization of any two arbitrary heaps, the efficient representation of allocation regions of unknown size (see section 5.2.2.), and the representation of arbitrarily many allocated regions within a representation of finite and bounded size (in order to allow guarantees to be made that a fixed point will be reached within some finite time during loop and function call abstraction even if iterations of the loop allocate dynamic storage: see section 4.3.5.2.2.). A concrete design has been developed to permit high quality modeling of heaps structures, and it is anticipated that this design will be incorporated into future modifications of the PARTICLE framework. Such high-level modeling of the heap will permit the code to be analyzed in a manner that captures the behavior of most programs far precisely would likely be possible using the low-level analysis discussed above.

5.2.4 Conclusion

This section has discussed the major design and engineering challenges remaining in the handling of the full C language and some of its crucial library routines. As mentioned above, PARTICLE makes a number of other minor omissions in its handling of the C language, and work must also be performed to extend the system to support such lacunae.

5.3 Reducing Resource Demands

5.3.1 Introduction

The compilation times associated with the current PARTICLE experiment are unacceptably expensive. The system requires large amounts of storage and can spend hours analyzing and specializing even relatively short programs. Although the quality of the analysis and optimizations performed by the current system is excellent, it is clear that the compilation overhead associated with the current experiment is far too great to allow practical application of the system. Preliminary and circumstantial evidence suggests that the speed of the current system can be increased by roughly a factor of five to ton by the introduction of more efficient data structure representations for abstraction and improved algorithms within performance-critical code. Improvement beyond that point will likely require considerably more thought. This section examines some of the more promising options for lessening the compile time and space overheads associated with the program specialization system; both incremental modifications and architectural changes are discussed.

5.3.2 Incremental Approaches

A number of evolutionary improvements to PARTICLE offer the potential to substantially reduce the overhead associated with compilation, without substantially decreasing the quality of program analysis performed by the system. Although the changes discussed in this section will not reduce compilation overhead to the level associated with traditional optimizing compilers, they may make it feasible to extend the the application of symbolic evaluation to a far larger and rich set of programs than that whose processing is currently feasible.

5.3.2.1 Adaptive Program Analysis

Previous sections of this document have discussed the importance of providing the user with fine-grained control over the richness of the abstract value domains and the precision with which program "power constructs" such as loops and recursive invocations are simulated during analysis. (See sections 3.3.4 and 3.4.). Such control permits users to focus high quality analysis on those areas of the program that require it the most while avoiding the squandering of compile time resources upon the analysis of regions of the program where such analysis is unlikely to be beneficial. Although the judgment of what constitutes worthwhile analysis rests ultimately upon the subjective preferences of the user and should always be open for explicit specification, there are a large set of cases in which PARTICLE could heuristically adjust the quality of analysis performed during symbolic evaluation in a manner that would be frequently be highly beneficial. Some ideas for possible heuristics are given below.

5.3.2.1.1 Adjustment of Domain Richness The height and size of the abstract domains manipulated during symbolic evaluation has a first-order effect upon the speed with which operations on values are performed, the space resources needed by symbolic evaluation and (most importantly) the time needed for abstracting program loops and recursive sequences. (See section 3.3.3.) Given the central importance of the size of the Value domains compile time resource consumption, it seems reasonable to consider a means of dynamically adjusting the height of that domain during symbolic evaluation in such a manner that those sections of the program or program quantities which seem most likely to require high quality analysis will be associated with large Value domains, while those portions of code or program quantities that have less need for precision during analysis will be associated with smaller and less precise but more space- and compile time- efficient Value domains. A few of the wide range of possible heuristics for adjusting the size of Value domains are described below.

A simple heuristic for adjusting the height value domain would be based on the extent to which the existing domain appears capable of representing values being computed. If the representation of the vast majority of existing values require only a small fraction of the expressive power of the domain, it is likely that the domain size could be restricted without unduly impacting the quality of analysis. Conversely, if there are an abundance of operations producing values which cannot be represented by the existing domains, it may be fruitful to enrich the domain representation to allow for the maintainance of higher-fidelity information regarding program values.[3] Explicit feedback systems regarding perceived representational needs allow

[3]Note that when such heuristics are applied separately to each program quantity and expression result and average over short periods of time, the results of the feedback begin to approximate systems in which there are *no* constraints upon domain size. When no cap is placed upon domain richness, the system automatically creates values of greater precision when needed; values that do not themselves take advantage of the full richness of program domains occupy no more space in systems allowing high maximal domain height than do such values in systems with low maximal domain height.

a means of low-pass filtering the demand for larger domains in either the spatial or temporal dimensions: Such heuristics adjust the domain richness in response to perceived needs, but only once sufficient evidence has accumulated that the richer representations are repeatedly needed, either by the contents of the same location (or the same expression result) at different points in time, or by several different values within the system. Feedback regarding domain sufficiency or over generosity be provided and applied at several different levels: For the program as a whole, for a the evaluation of a given function or construct, or for a particular program quantity or temporary result.

There is another simple metric that can be used to flag cases in which the existing domain size seems likely to be unnecessarily generous. Rather than attempting to judge the need for a representation on the basis of characteristics of its *use* (as was done by the heuristics presented above), however, one can look instead at the degree at the frequency with which the symbolic evaluator has recently been discovering possible specializations: If a certain Value representation has permitted the realization of a number of different specializations, it seems likely that restricting that specialization might damage the quality of analysis performed by the same; by contrast, if a rich Value representation has not recently permitted any specialization, it seems plausible to conjecture that it is not necessary to maintain the full richness of that representation. While it is *possible* that maintaining rich representations at some point during symbolic evaluation will provide the basis for discovering optimizations only at a much later time (e.g., after the return of the function call being evaluated), the high temporal locality exhibited by most programs seems likely to make this somewhat rare (particularly when optimizations are considered over a sufficiently large window of time). The use of such a heuristic can be particularly helpful when used to accelerate the analysis of pieces of program code in which the analysis takes advantage of the rich domain by maintaining very large sets of Values but consistently detects no opportunities for specialization.

Another heuristic for adjusting domain richness could make use of information regarding the frequency and importance with which a given program quantity or its logical "descendants" is manipulated within the program text or has appeared previously within symbolic evaluation (either within the current invocation of the surrounding function or within previous evaluations of that function). Using such a heuristic, program quantities which are manipulated extensively or are used within particularly important operations (e.g., control flow, function calls, in pointer operations or as loop indices, etc.) might be allocated particularly generous domains in order to allow the symbolic evaluator to capture more precise information about the possible values associated with such quantities. On the other hand, program values that are used little or in a relatively innocuous manner would tend to be granted smaller abstract domains. Simple semantic information concerning the contexts in which program quantities appear can help the program specialization system judge the probable importance of precisely simulating such quantities, and can aid in the selection of appropriate abstract domains for those quantities.

A final heuristic for dynamically adjusting the size of the Value domains would be based upon a perception of the likely benefit specialization of the surrounding piece of code would offer at compile time. In particular, the symbolic evaluator could allocate increasingly less precise and full-bodied domains as the execution of a given piece of code is judged as increasingly speculative in accordance with the philosophy dictating that compile time resources should be applied to code in proportion to the expected run time benefits associated with optimizing that code. Within such a system, code nested deeply within a series of conditionals associated with unknown predicates will receive less compile time attention and will be analyzed using simpler abstract domains than code directly on the main path of execution of a program.

Each of the heuristics described above seems likely to offer some promise for accelerating program analysis by concentrating the use of rich abstract domains where they are most needed and avoiding the costs attended upon the use of such precision where it offers no performance advantages. Substantial experimentation is needed to test and refine the methods described and to explore other heuristic strategies. The next section turns to consider the adaptive throttling of another highly powerful but expensive component of the analysis carried out by PARTICLE: The fine-grained simulation of loops and recursion.

5.3.2.1.2 Throttling of Control Flow Simulation Given that the primary goal of the program specialization explored by PARTICLE is performance enhancement, it would be ideal for the program specializer to shift as much run time computation to compile time as possible within the limits imposed by the user on specializer execution time. Unfortunately, it is not only practically but also logically impossible for the program specializer to distinguish all cases of useful compile time computation from instances in which symbolic evaluation offers no performance benefit. Nonetheless, there are a wide variety of heuristics that can be applied during symbolic evaluation process in order to estimate the importance of simulating a given computation in a fine-grained manner. This section discusses the applications of heuristics for throttling a small but particularly important and time-consuming component of symbolic evaluation: The simulation of loops and recursion (hereafter collectively referred to as "loops"). Section 3.4 discussed some of the challenges involved in implementing a powerful heuristic loop and recursion termination strategy: Although the need to guarantee specializer termination requires that the simulation of all program loops and recursive sequences must terminate in some finite time, in order to shift maximal computation from run time to compile time it clearly seems desirable to allow computations judged as "useful" or expensive to be simulated for far longer than those deemed "useless" or less costly. The key difficulty lies in the formulation of good heuristics to estimate the desirability of compile time computations. As was the case for adjusting domain richness that were discussed above, some of the best heuristics for throttling loops are adaptive, using feedback from the symbolic evaluator to guide the decision as to whether to continue fine-grained loop simulation or to begin the process of loop abstraction. (Note that the methods described below could also be used to allow for a more graceful degradation in the quality of loop analysis between fine-grained simulation and abstraction. Thus the methods presented below could be combined with those presented in the previous section as means of adaptively varying the size of program domains. Such a strategy could operate by making the abstract domains and the control flow analysis within fine-grained loop simulation increasingly less precise, until it is judged that abstraction of the loop is needed.) Some preliminary ideas for loop throttling heuristics are given below.

The ultimate justification of all program analysis within PARTICLE lies in the value of the specializations permitted by the results of that analysis. Section 5.3.2.1.1 considered the potential for judging the benefits of a given domain size by considering the number and importance of specializations permitted by the use of that domain. A very similar metric can be used to judge the usefulness of simulating a loop at compile time. The adoption of such a strategy is based upon the observation that without large amounts of specialization within the loop itself, much of the loop will probably also have to be executed at run time, and, therefore, that there is likely to be relatively little direct benefit to be gained by executing that loop at compile time. The lack of specialization also serves as a secondary indication that there is unlikely to be much precisely known static data concerning the program quantities manipulated and processed upon by the loop. Unfortunately, while the use of this metric is appealing for a wide variety of loops, it gives poor results for a restricted subclass of loops: Instances such as initialization loops which manipulate but do not *compute* statically known data. Processing such loops will frequently not reveal in any possible specializations, but will provide the symbolic evaluator with substantial amounts of knowledge about program data – knowledge that often permits the discovery of specializations at later points in the program analysis. The next heuristic attempts to capture most of the the benefits of the current strategy while avoiding the nearsightedness that can result from simply counting the specialization permitted within some window of time.

The discussion above has already alluded to the fundamental difficulty associated with a heuristic throttling strategy based upon counting and weighing the importance of specializations discovered during loop execution: In certain cases, while the fine-grained simulation of a particular piece of code may not reveal opportunities for performing any specializations *within the processing of that code*, the precise analysis of that code may collect sufficient static information to allow specializations later in the program. Rather than judging the usefulness of a loop by counting the number of specializations permitted *during the analysis of the loop*, it will frequently be more revealing to consider the degree to which the loop provides a basis for possible specialization. Because this information is not available when processing the loop itself, it is necessary to find some appropriate approximation that can be calculated "online" during loop evaluation. Given the fact that the ability to specialize program code frequently depends directly on the precision of knowledge concerning the data manipulated within that code, a crude but obvious approximation to the likely value of the loop analysis could be found by considering the precision of the data manipulated within that loop. If the loop is operating upon poorly known data, it is highly unlikely that the data being manipulated would permit specialization either within the body of the loop itself or within subsequent code. Conversely, if the loop manipulates a great deal of precisely known data, it is quite possible that the data will at some point open opportunities for performing important specializations. As a result, the precision with which values are known within a program loop can frequently offer a reasonable amount of information about the chances that the careful simulation of the loop will permit substantial optimization, and could thus be used as a metric for estimating the benefit to be gained by the high-fidelity simulation of a particular loop. It seems likely that a more powerful and intelligent strategy yet could result from combining this metric with an earlier heuristic and permitting the creation of a throttling strategy that makes use of knowledge about *both* the overall importance of specializations already permitted within the loop and about the precision of the data being manipulated. Information concerning previously discovered specializations will allow some concrete knowledge about the benefits already allowed by the character of the data manipulated within the loop, while information about the precision of that data will allow some means of for estimating the likelihood that additional specializations will be permitted at some later point.

PARTICLE currently makes use of a small amount of feedback from symbolic evaluation in order to throttle loop simulation: The system provides different handling for loops associated with a termination condition of known truth value and those loops whose termination condition evaluates to an unknown truth value. As discussed in section 3.4, however, such local criteria provide only a crude means of approximating a more important statistic: The estimated overall likelihood that a given iteration of the loop body being simulated will *actually* be executed at run time (a statistic that is itself an estimate of the likelihood that a given specialization discovered during symbolic evaluation will actually be beneficial at run time). A more precise estimation of this likelihood can be arrived at by considering the overall "speculative nesting" of the loop body to be simulated – the number of unknown conditionals in which the loop body is *dynamically* nested.[4] This information could be used to gradually constrict the degree to which fine-grained simulation is used to process loops bodies whose run time iteration is regarded as increasingly unlikely. The same statistic could be used to adjust the quality of value analysis performed during symbolic evaluation.

5.3.2.1.3 Conclusion This section has examined the prospects for using of feedback from the symbolic evaluator to dynamically adjust the precision of program analysis. The fundamental motivation for the heuristics described above is the desire to concentrate compile time resources upon those regions of the program in which high quality analysis likely to yield the best results, and to avoid the overhead associated with such analysis when it is not needed. Feedback from the symbolic evaluator can provide important hints as to how valuable a certain compile time evaluation is likely to be at run time, but there are other sources of information that provide additional clues as to the importance of that evaluation. The next section examines another basis for heuristics designed to throttle the quality of program analysis.

5.3.2.2 Profile-Guided Program Analysis

The discussion in the previous section discussed the use of adaptive heuristics for throttling symbolic evaluation and suggested a means of adjusting the precision of analysis for a given piece of code based upon estimated frequency with which that code would be executed at run time. This heuristic attempted to derive the frequency of code execution based upon knowledge of the speculative depth of the construct being symbolically evaluated. An alternative approach to estimating execution frequency information would rely on fine-grained profiling data collected from execution of the program on representative input.[5] Execution frequency data for a given piece of code could be used to adjust the size of the abstract domains used during the symbolic evaluation and abstraction of that region and to set the point at which loop abstraction is initiated within that code (loops whose execution is likely to be short could be iterated directly, while loops with large numbers of expected iterations might begin abstraction immediately).

The value of all profiling information is predicated upon its ability to accurately reflect the patterns of run time behavior seen for "representative" input. For some programs, the

[4]Note that it seems likely to the author that would prove unrealistic to consider each iteration of a loop associated with an unknown termination condition as executing within an additional level of unknown conditionals. In general, it seems likely that a given loop termination condition is far more likely to be false than true, while the average conditional may be associated with a far less skewed distribution. To be most realistic, each speculative reliance upon a loop termination condition should be probably be weighted as a *partial* reliance upon an unknown conditional

[5]The profiling information used by the symbolic evaluator would ideally include frequency information on the execution on each statement for each dynamic *call* of that function. Such information would enormously improve the precision of the symbolic evaluator's judgments of the run time execution frequency of a given statement *for a particular call.*

selection of characteristic input can be difficult and it may consequently be difficult to devise an appropriate profiling strategy. Even in such cases, however, it seems likely that the use of profiling information may offer substantial analysis advantages as a supplement to other heuristic means for approximating code importance.

5.3.2.3 Removing Interpretive Overhead

Previous sections have discussed methods for improving PARTICLE's symbolic evaluation strategy in order to allow a more fruitful allocation of scarce compile time resources. Another approach would seek simply to decrease a substantial but unnecessary overhead associated with the current techniques for symbolic evaluation. As discussed in section 4.3.2, symbolic evaluation evaluates program constructs by walks over their abstract syntax representations. The computations underlying such walks involve substantial pointer-following are associated with a large set of memory references during symbolic evaluation, and in general seem likely to be associated with substantial performance overhead. Because the structure of the input program is completely known at the beginning of symbolic evaluation, the input program could be compiled internally into an intermediate language consisting only of calls the PARTICLE framework (memory management and value manipulation routines, etc.). The processing of such a compiled program would eliminate the interpretive overhead associated symbolic evaluation and increase locality of reference during symbolic evaluation, and opens the possibility for substantially enhancing the compile time performance of PARTICLE.

5.3.2.4 Opportunities for Parallelization During Symbolic Evaluation

As discussed in section 4.3.3.2.1.2, the handling of unknown patterns of control flow within PARTICLE requires constructing legal approximations to the effects of *all* possible control flow paths. The process of creating a valid approximation to the effects of any of several different possible paths of execution requires requires the independent simulation of each of these different paths and a merging of the resulting states. There are never any data dependencies between the independent paths to be simulated, and in many instances the amount of computation required to symbolically evaluate each possible path can dwarf the overhead associated with forking off independent threads for those paths. The symbolic evaluation of possible paths of execution is thus eminently well suited for parallelization. Given the amount of unknown control flow within practical programs, it seems likely that parallelizing the symbolic evaluation stage of PARTICLE could dramatically accelerate program analysis.

5.3.2.5 Memoization

The FUSE symbolic evaluation system [56] made use of "function caches" which memoized the return values and specializations associated associated with function calls, and permitted reuse of these cached results during later symbolic evaluation whenever judged safe and desirable [56][46]. While the function cache within FUSE was emphasized primarily as a means of allowing reuse of *specializations* rather than as a technique for decreasing the overhead associated with symbolic evaluation, [46] reported that use of the function cache decreased the running time of symbolic evaluation by an order of magnitude.

There are a wide variety of opportunities for memoization during symbolic evaluation, and although PARTICLE takes some advantage of those prospects during expression evaluation (see section 4.2.2.1.), the current system makes no attempt to exploit the more lucrative potential for memoization at higher levels of symbolic evaluation (e.g., by caching the state mappings performed by function calls or statements).[6]

While straightforward memoization of input/output state pairs for higher-level constructs would allow some opportunities for decreasing symbolic evaluation overhead, such an approach is unnecessarily restrictive and would greatly constrain the opportunities for reuse of memoized entries: In virtually all instances the construct will rely on relatively small components of the input state and will modify a small fraction of that state itself. Rather than imposing cache reuse conditions which require that the simulated state at the point of reuse be identical to the input state associated with the cache entry, looser conditions can be imposed in which the input state need only correspond to the cached input state for those components of that state which were relied upon during the memoized execution of the construct.

Even in those regions of the input state whose values are used during the evaluation of the construct, in order to reuse the memoized entry it is not always necessary for the input state and and memoized input state to match *exactly* – for instance, it is possible that the output state for a given function call will be the same if a set of parameters are associated with the same values and the final parameter is any integer greater than zero. In such a case, it would be legal to reuse a given memoized input/output mapping without the last parameter at the point of reuse to *exact* match the corresponding parameter in the memoized input state – all that is required is that the values associated with that piece of memory in each state are both positive or both negative. In general, a given input/output mapping can have arbitrarily complex conditions for legal reuse (indeed, the question as to whether a given cache entry can be reused is in general uncomputable). There are a variety of compromises possible that strike different balances between the expressiveness of the reuse conditions and the aggressiveness with which caching will be performed against the time necessary to check reuse conditions and overall system complexity.[7] The selection of a maximally desirable strategy for capturing memoization reuse conditions would ideally be empirically based upon observation of the time savings associated with different possible reuse algorithms when operating on realistic code.

Reuse conditions at the very simple end of the expressiveness scale might insist that any point of reuse for a memoized input/output pair be associated with a state approximated by the input state for the memoized pair in all locations which are accessed during the execution of the memoized code.[8] Such a condition can be rapidly checked and compactly expressed,

[6]For all but the coarsest forms of memoization, it is impractical to maintain a complete cache of input/output mappings for a given construct and some means of discarding mappings must be used. It seems likely that standard LRU or pseudo-LRU replacement strategies would work well in such conditions, although more sophisticated schemes could be used.

[7]It should be stressed that previous symbolic evaluation systems [46] have used input/output memoization both for decreasing the compile time cost associated with symbolic evaluation and for flagging opportunities for reusing previously discovered specializations; within such systems, memoization played a crucial role in discovering the opportunities for performing *any* specializations. PARTICLE enforces the reuse of specialization through alternative and less restrictive methods; memoization within PARTICLE would thus serve only as a means of increasing the speed of the specialization process.

[8]For simplicity of exposition, the current discussion is glossing over several important aspects of reuse conditions: In general, what needs to be matched between the input template for an input/output cache entry and

but (as discussed above) is not capable of allowing memoized entry reuse in a wide variety of circumstances.

More sophisticated representations might allow limited use of expression-based constraints on program quantities or locations within the input state. Such representations would permit expression of reuse conditions such as $v > 0$, $*p + v < 100$, where p and v are program quantities in the entry state to the memoized code. Representing and checking such conditions would be more costly than for the simple knowledge matching scheme proposed above, but such a scheme might permit significantly greater precision in specifying conditions for reusing a given memoized piece of code and thus significantly lower the overall cost associated with symbolic evaluation by allowing reuse of a substantially larger set of memoized entries.[9]

Unfortunately, conditions under which it is safe to reuse a particular memoized input/output mapping can be arbitrarily complex and in general the expression of those conditions requires computationally universal predicates. The conditions given above all described contexts in which it is legal to reuse a memoized state/state mapping in terms of quantities found in the input state. The reuse conditions given as examples above were fairly simple, but such conditions can also depend upon complex computations performed within the function to be called. As a somewhat contrived example, consider a routine which accepts an integral argument and an array floating point numbers. This routine first checks to see if the argument is a prime number. If the argument is prime, some expensive sequence of operations is performed upon the array. If the argument is composite, another different but equally expensive set of operations is used to process the input array. Suppose now that the symbolic evaluator has simulated the evaluation of one such call to this procedure with a known prime integral argument and exactly known array contents, and wishes to memoize the input/output mapping for that call. It is not immediately clear what reuse condition to place upon the value of the integral argument: It would be ideal if the symbolic evaluator could recognize that any future call site with an integral argument known to be prime and with the same array contents could make use of the earlier memoized results. But this requires some manner of expressing a reuse condition requiring that the argument is prime. In general, if the fundamental operation of a function hinges strongly on the results of some preliminary computation performed within that function, the reuse conditions for a given memoized mapping will also depend strongly upon the results of that preliminary computation (e.g., the computation to check if a given argument is prime) – any other reuse conditions will be unnecessarily restrictive (e.g., requiring that the integral input parameter is associated with *exactly* the same value as in the earlier call in order to make use of the earlier memoization.) In order to permit the use of such reuse conditions, a universal representation must be adopted, permitting the use of arbitrary code in the conditional predicates. Such conditions are capable of exactly specifying all conditions for reuse of a function. Unfortunately, their universal character permits expression of condition checks

the state at point of desired reuse is *knowledge about program quantities.* In particular, the input template must represent a legal approximation to the corresponding aspects of program state at the point of reuse. This approximation must extend to any knowledge of equivalence between values that is based upon equivalence identity tags (see section 4.2.2.1.) that is relied upon by the memoized execution being considered. (Thus, if two values in the input template are known to be equivalent by virtue of possessing the same identity tags, those values must possess the same identity tags wherever the memoized input/output mapping is reused.)

[9]Note that because the reuse of a specialization must be based on *absolute* knowledge that the condition applies, by allowing greater *precision* in specifying the reuse condition, the system can allow much more *general* reuse of the memoized input/output pair.

of arbitrarily computational complexity. Moreover, checking different complex conditions for the legal reuse of a given memoized input/output state pair may lead to repeated reexecution of much of the same code predicate code – leading to a situation in which checking the reuse conditions for a memoized pair is more expensive than simply evaluating the associated code whose mapping was memoized. For instance, predicates expressing reuse conditions for a function may involve internal values calculated deep within that function. In order to express such conditions, the system will be forced to include the preliminary code involved in calculating the values tested within the reuse predicates. Without non-trivial bookkeeping machinery, a given piece of preliminary code may be duplicated within several different reuse predicates, and be maybe executed at many points during the checking of reuse conditions for one or more memoized entries. If during symbolic evaluation the system is required to check a large number of reuse predicates, checking the opportunities for reuse of a memoized entry may prove more expensive than reevaluating the function whose evaluation was memoized.

In [46], Ruf and Weise describe a technique for parameterizing memoized return value specifications for function calls by the argument specification for those calls. This parameterization permits the reuse of a given memoized pair expressing the return value for a function for a wide variety of different arguments, even if the return value of that function directly refers to the *particular* values of those arguments. This parameterization permits the reuse of a memoized pair under a wider variety of conditions than would otherwise be possible, and may offer even greater advantages in the memoization of input/output maps for imperative languages. The conditions for simple parameterization are easily computed and can be efficiently implemented.

This section has reviewed a range of possible strategies supporting the memoization of the results of symbolically evaluating a given piece of code. Memoization can save significant compile time by allowing the symbolic evaluator to avoid needless re-simulation of program behavior. Memoization permits the symbolic evaluator to exploit the substantial degree of spatial and temporal locality within statically observable patterns of program behavior with no negative impact upon the quality of specializations performed by the system. Simpler memoization schemes often impose grossly restrictive reuse conditions, but support highly efficient checking of such conditions. More general schemes allow expression of less restrictive conditions, but the condition checks impose a higher computational overhead in checking those conditions. Reuse conditions which employ universal reuse predicates can exactly capture the conditions under which it is safe to reuse a given memoization, but can lead to situations in which the checking the conditions for the reuse of a particular memoization is more expensive than evaluating the code the result of whose symbolic evaluation has been memoized. In all likelihood, some combination of such schemes would offer the most promising memoization strategy: The system could make use of more powerful schemes where judged appropriate, but could fall back on more efficient but restrictive strategies where it seems likely that precise specification of reuse conditions would be prohibitively expensive. Parameterized memoization schemes represent a simple and efficient way of broadening the conditions under which a memoization can be reused under any of the schemes considered, and could be incorporated within any memoization framework.

5.3.2.6 Conclusion

This section has sketched a number of modifications to PARTICLE that could be used to accelerate symbolic evaluation. All of the changes presented operate within the current analysis framework and conceptually represent minor modifications to PARTICLE. Were several of such changes adopted together, it seems likely that the time required by the symbolic evaluation phase of PARTICLE could be decreased by at least two orders of magnitude. Unfortunately, it seems likely that even with such performance PARTICLE would be unable to realistically process many large software systems. The next section turns to consider more fundamental changes to the current PARTICLE system which would allow effective processing of even the largest software systems.

5.3.3 New Techniques based on Preliminary Program Analysis

Just as the information collected during program analysis can reveal opportunities for improving the run-time performance of the program being analyzed, opportunities for improving the performance of the *symbolic evaluation* of a program may be revealed by the appplication of less exact analysis to that program. Data collected within such an analysis prepass can vary widely in precision in character (from simple information about characteristics of the program text to patterns revealed by full-fledged traditional interprocedural data flow analysis), and later phases of the analysis and symbolic evaluation itself can make use of and sharpen the results of earlier phases.

Although the information collected through pre-symbolic-evaluation analysis can be put to use in guiding symbolic evaluation in many different domains, this section is limited to considering two of the most important applications of such knowledge. In particular, simple preliminary program analysis can help accelerate the two most computational expensive aspects of symbolic evaluation: The simulation of function calls and loops. The use of such analysis in improving performance in each of these areas is sketched below.

5.3.3.1 Throttling of Function Calls

The fundamental difficulty with the program analysis algorithm currently employed by PARTICLE lies in its utter disregard of function boundaries. Aside for exceptions due to the presence of cycles, PARTICLE is guaranteed to perform a depth-first walk over the call graph that permits reexploration of any given node of the graph an arbitrarily large number of times during the exploration process. Conceptually, the exploration of the call graph is isomorphic to an unrestricted depth-first exploration of a DAG. The number of function calls involved in such an exploration can be proportional to an exponential of the number of functions within the system (where the exponential constant is related to the fan-out associated with each node). As a result, analysis within the current framework is hopelessly impractical for all but relatively small software systems. Without curtailing the willingness of PARTICLE to explore all paths through the call graph it seems doubtful whether the system can ever be fashioned into a practical software tool. On the other hand, one of the great strengths offered by PARTICLE lies in its very ability to thoroughly exploit the substantial amount of static information available at function boundaries, particularly by virtue of symbolic evaluation of calls associated with large amounts of static data (See section 2.4.2.2.2.2.). This section sketches the fundamentals basis for a system designed to permit the discovery and thorough exploitation of such interprocedural information in an analysis framework far less expensive than that currently implemented in PARTICLE. In accordance with the general philosophy of PARTICLE, the framework permits the user fine control over the quality of the analysis performed by the system and the amount of time required by that analysis.

Within the current PARTICLE experiment, each function *f* is evaluated whenever called from a predecessor function. This function evaluation may involve the simulation of further calls within *f*, and is followed by the return of an approximation to the function termination state. The policy of explicitly simulating all function calls permits the symbolic evaluator to simultaneously compute a maximally precise approximation to the return value of a function, calculate of the effect of a function call on global program state, and discover opportunities for specialization within the called function. Just as high quality iteration-by-iteration (or recursion-by-recursion) loop simulation permits PARTICLE to approximate the effects of program loops far more accurately than is possible within existing compilers, high quality simulation of function calls allows the system to exploit far more information across procedure boundaries than is possible within even the most aggressive optimizing compilers. Unfortunately, while PARTICLE provides flexible and effective means of throttling the simulation of program loops, the system currently offers no manner for trading off precision and compile time in the simulation of function calls. Instead, all function calls are simulated in a highly precise but enormously expensive manner. It seems natural to consider extending the current system to allow the application of less expensive function call strategies where desired.

Just as the current system allows the explicit simulation of program loops to proceed for a limited number of iterations before such loops are analyzed in a far faster but less precise manner, a generalized system could restrict symbolic evaluation to a behavior in which the system would only perform explicit simulation of some specified number of nested calls before approximating the results of such calls in a far more coarse-grained manner. While the simulation of each nested call before the limit would be performed in a full-bodied manner, any call to a function which exceeded the specified limit on dynamic nesting of function invocations would simply record the current system state at the entry point to that function and return some highly conservative estimate of the return state and value of the called function. At some later point in the execution of the system, the system could could LUB together all pending call states for a given function and begin independent simulation of that function (perhaps after again resetting the count on the number of function calls to explicitly simulate). In this manner, paths of function invocations which exceed some specified length would be generalized together before continuing – preventing the tremendous fan-out of paths of symbolic evaluation seen within the current system. (Such a technique corresponds conceptually to an abandonment of simple depth-first call graph exploration once a certain depth has been reached, replaced by a transient shift to a breadth-first strategy.)

While a strategy imposing a simple limit on the depth of full-bodied function call simulation would be effective in greatly curtailing the expense associated with function calls, such a policy effectively positions an arbitrary barrier to high-quality analysis in a manner that is completely insensitive to the patterns of underlying data flow within the program, and thereby also deprives the system of important opportunities for thorough symbolic evaluation in those cases where a great deal of static information is available. Substantial improvements to such a strategy can be obtained by giving the system access to knowledge concerning the likely costs and benefits associated with making a particular function call. In particular, a preliminary analysis could collect information estimating the potential benefits of static knowledge about particular program quantities at different points in the code (intuitively, estimating the "importance" of knowledge about such quantities at such points), and the likely cost of the symbolic evaluation necessary to reap the benefits of such knowledge. Armed with such information, the system can attempt to judge the magnitude of the benefits that would result from the symbolic evaluation of particular function calls (based upon the amounts of static information available at such calls), and could attempt to balance such expected benefits with some perception of the compile-time costs associated with the full-bodied symbolic evaluation of the called function.

The collection of the information necessary to permit the throttling of function calls in this manner is relatively straightfoward: A prepass independently processing each function could collect information concerning the usage and importance of program quantities used within that function and for each such quantity give some estimate as to the compile-time cost associated with symbolically evaluating from the beginning of the function to the point or points where such information is used. A set of passes over the call graph then be used to find the interprocedural transitive closure of such information, thereby allowing estimation of the relative magnitudes of the benefits expected from knowledge of particular program quantities at program call sites and the the compile-time cost required to exploit static knowledge of such program quantities from that call site. (Performing transitive closure over the call graph is a standard procedure within traditional interprocedural analysis, while the use of program analysis to find of "important uses" of program quantities is similar to the techniques used to throttle the propagation of cloning vectors within [11].) During symbolic evaluation itself, calls could be simulated where information from the preliminary analysis indicated that sufficient important static information is known to make worthwhile the estimated compile-time cost needed to exploit such information.

Judging the "importance" of a particular program quantity in a function will require the use of empically test heuristics, likely involving factors such as the estimated frequency of usage of that program quantity within that function, and the *manner* in which the quantity is used (e.g. the use of a program quantity as an index and pointer variables or in the direction of program control flow might be judged to lend greater importance to that quantity.)

Despite the need for a separate analysis pre-pass, the ability to constrain the simulation of function calls by appeal to information concerning the likely cost/benefits of such simulation seems likely to provide an powerful, flexible, and straightforward manner of enormously accelerating program analysis while maintaining the considerable bulk of the benefits of full-bodied symbolic evaluation. Moreover, the modifications necessary to support such a system could be added to the current PARTICLE framework with a minimum of difficulty. It is expected that the adoption of such a methodology would represent a crucial step towards making the application of PARTICLE practical for large software systems, and could be accomplished without any serious difficulties.

5.3.3.2 Accelerating Analysis of Loops

One of PARTICLE's greatest strengths lies in its capacity for high-quality loop simulation. During both iteration-by-iteration and fixed-point loop analysis the system has the potential for approximating the effects of program loops considerably more accurately than is possible using traditional loop analysis strategies. Because programs tend to spend large fractions of their time iterating within program loops of rather small extent[33], such detailed loop analysis can offer substantial improvements in the run-time performance of the code being optimized. At the same time, because the time spent within loops dominates the run time of most programs being analyszed, it is desireable to make the symbolic evaluation of program loops as rapid as is possible without substantially sacrificing the accuracy of the analysis.

Considerable economies in the cost of loop simulation can be gained by exploitng the fact that it may not be desireable or even necessary to reexecute each statement within the loop for each iteration of that loop. In particular, it may be that symbolic evaluation will reveal that the results of certain expressions within the loop are demonstrably unchanging for particular iterations of the loop[10] or have only changed in manners which it is safe and cost-effective to ignore (e.g. if the inputs to an expression in a particular loop iteration are conservatively approximated by the inputs within the previous loop iteration or if the previous resulting value of the expression were $\top$, then the previous output of the expression is guaranteed to represent a conservative approximation to any new possible output of that expression). In such cases, it may be desireable to avoid recalculating of expression results and reuse the old output value. Similarly, if a particular expression computes a value that is deemed "unimportant" by (flexible) criteria similar to those discussed within the previous section, it may be worthwhile for the system to avoid any attempt at evaluating that expression and to simply *always* approximate the output of that expression as $\top$.

By systematically exploiting such opportunities for eliminating needless computation during loop fine-grained and fixed-point loop iteration, it seems likely that the static analysis of loops within PARTICLE could be significantly accelerated. Such effects could be particular pronounced during fixed-point iteration, in which many values are likely to reach fixed points (and particular $\top$) prior to the convergence of the entire loop, and where the practice of approximating values judged to be unimportant as $\top$ could yield much more rapid convergence than might otherwise be the case. An analysis prepass is not strictly needed to make use of an algorithm which avoids much needed computation, but information gained through such a prepass may be able to substantially lessen the amount of bookkeeping needed during symbolic evaluation, and (more importantly) such a preliminary analysis pass also offers the opportunity for collecting information on the estimated importance of static knowledge concerning particular program quantities. (See previous section). The integration of such an importance metric into judgements about when it is worth simulating a loop statement opens the opportunity for creating a highly flexible loop analysis strategy which allows the user to choose a careful balance between the cost and precision of compile-time analysis. In particular, by setting the metrics by which the "importance" of program quantities are judged and the threshold at which the calculation of a program quantity is considered sufficiently important to simulate, the user can exercise fine control over the precision of loop analysis.

While the modifications discussed above offer the opportunity for significantly accelerating the speed of loop analysis, it seems likely that such changes to the current analysis strategy can be made relatively easily within the current framework. There is significant choice available in selecting the exact mechanisms used to implement the variations suggested above – for example, forms of memoization could be used to discover some opportunities in which reexecution is

[10]Note that expressions need not be loop-invariant in the traditional sense in order to be invariant for a *particular* iteration of the loop executed during fine-grained symbolic evaluation. That is, while such expressions may depend on program quantities that are changed within paths of execution taken at *some* point during the loop execution (and thus fail the standard criterion for loop invariance), the precise symbolic evaluation of the loop may indicate that *during a particular iteration of the loop* such quantities remain unchanged.

judged unnecessary, while other techniques might depart more substantially from the symbolic evaluation framework and make central use of a worklist-based fixed point algorithm similar to that presented in [4]. It remains to be decided what mixture of techniques will be most appropriate for use with the types of static information guiding statement execution within loops.

5.3.3.3 Concusion

The sections above have presented two examples in which information collected by a simple analysis pass over the program prior to symbolic evaluation could be used to significantly (and, for the function call example, drastically) accelerate the speed of program analysis. Suprisingly, such performance improvements can be achieved with only moderate additions to the current symbolic evaluation framework. The ability to achieve such substantial decreases in compile time without significantly sacrificing the quality of program analysis or the need to make substantial alterations to the current PARTICLE framework makes the adoption of the techniques discussed above highly desireable, and such modifications will likely within the near future.

5.4 Reducing Sensitivity to Slight Perturbations in Program Phrasing

Even when coupled with rich abstract domains, the analysis performed by PARTICLE can be surprisingly sensitive to small details in the manner in which programs are expressed. Figure 5-3 illustrates a case in which slightly different manners of performing the same computation can lead to drastically disparate results during program analysis. Both pieces of code compute the factorial function in an iterative manner for an argument n which is either 5 or 6 but the quality of the program analysis performed for each of the two cases varies widely.

In the first code fragment, the counter i associated with the factorial begins at n and counts down to 1. Because i begins at n and counts downwards, each successive value of i is imprecisely known. Because at each step of the computation the current value of i is accumulated as part of the running factorial total, with each iteration that total becomes increasingly less well known. By the point at which the loop is known to terminate, the total is at best known only to be one of a wide range of possible values.

In the second piece of code, i begins at 1 and counts up to n. Because i begins at 1 and is successively incremented by the code, i is fully known for each iteration of the loop. Because n is not known, the symbolic evaluator does not know if the loop will execute 5 or 6 times, and is thus forced to create an approximation to the termination state of the loop in which the total is known to be either 5! or 6! At no point within the symbolic evaluation of this fragment does knowledge about the final answer experience the successive dilution seen within the first fragment; the final answer is known as precisely as possible given the available information about the value of n. The two examples shown in figure 5-3 are remarkably similar in structure, and the fact that analysis of the the code produces remarkably different results for the two cases seems a cause for considerable concern.

Figure 5-5 presents another example in which minor changes in the implementation of a given algorithm can lead to even more drastic changes in the quality of the analysis performed by the system.

```
{
        /*  n known to be either 5 or 6 */

        i = n;
        total = 1;

        while (i > 0)                      /*  i NEVER fully known */
                {
                /*      ignorance about the value of total grows with
                        each iteration of the loop */

                total = total * i;
                i = i - 1;
                }
}
```

Figure 5-2: The symbolic evaluator is unable to obtain a precise value for *i* at *any* point during the analysis of the program fragment above. As a result, the imprecision associated with the Value of *total* grows rapidly and yields a final Value for *total* that is wildly inexact.

```
{
        /*  n known to be either 5 or 6 */

        i = 1;
        total = 1;

        while (i < n)                      /*  i ALWAYS fully known */
                {
                /*      the value of total within the loop is always
                        fully known
                */

                total = total * i;
                i = i + 1;
                }
}
```

Figure 5-3: Throughout the analysis of the *while* loop, the symbolic evaluator is always able to maintain exact knowledge of the values of *i* and *total* associated with each iteration. The sole uncertainty within the analysis concerns the issue as to whether the loop is executed 5 or 6 times. At the end of the loop, *total* is known to be associated with either 120 or 720.

```
int  fact(int n)
{
        if (n == 0)
                {
                return(0);
                }
        else
                {
                return(n * fact(n - 1));
                }
}
{
        ...

        fact(k);                /* k known to be either 5 or 6 */
}
```

Figure 5-4: The symbolic evaluator is unable to obtain a precise value for n at any point during the analysis of the recursion above, and thus the condition $n == 0$ is thus never known to be true. As a result, the recursive sequence is simulated until the abstraction process is begun.

```
int  fact(int n)
{
        if (n <= 0)
                {
                return(0);
                }
        else
                {
                return(n * fact(n - 1));
                }
}
{
        ...

        fact(k);                /* k known to be either 5 or 6 */
}
```

Figure 5-5: Although the exact value of n is never fully known during the analysis of the fragment above, the $n <= 0$ predicate is known to be true after the simulation of 6 recursive invocations of the function. As a result, the system avoids the need to perform abstraction upon the recursive sequence.

Consider the invocation of the first code fragment with an argument n which is known to be 5 or 6. During each successive recursive application of the function n is decreased, and the predicate $n <= 0$ is known to be false until n is associated with a value of either 0 or 1. At this point, the symbolic evaluator will speculatively execute a recursive call, n will be decreased one more time and will then be associated with the possible values -1 or 0. As a result, the predicate $n <= 0$ will be known to be *true* and the chain of recursive invocations will be broken. Because each n associated with a recursive invocation is known only to be one of two values, the value returned by the chain of recursive invocations will be very poorly known (as was the case in the first example of figure 5-3.); unfortunately, the analysis of the second recursive factorial example exhibits even worse behavior.

Within the second code fragment in figure 5-5, the symbolic evaluation of the recursive sequence will begin just as was the case within the first piece of code. The analysis will be precisely the same until the point at which n is known to be associated with the value -1 or 0. Within the first piece of code, the value of n is tested within the predicate $n <= 0$, which is known to be true for this value of n. Unfortunately, in the second fragment of code, this predicate is replaced by the predicate $n == 0$. While knowledge that n is either -1 or 0 is sufficient to determine that $n <= 0$ (within the first fragment), such knowledge is *not* sufficient to determine that $n == 0$ (in the second fragment). As a result, while the first fragment of code was able to definitively return and terminate the sequence of recursive invocations when n is equal to -1 or 0, the second fragment must continue by speculatively executing both branches of the *if* statement. Processing one of these branches leads to another recursive invocation of the function with an argument known to be either -2 or -1. During this deeper invocation, the predicate $n == 0$ is definitively known to *not* be true – leading to the evaluation of a series of recursive function calls associated with *known* recursion conditions. During symbolic evaluation, PARTICLE would eventually be forced to terminate the series of calls and begin abstracting the sequence of recursive calls – a process likely to yield a *grossly* conservative estimate of the possible return values of the original call.

The two pieces of code in figure 5-5 differ only in the choice of relational operator used to check for the base case for the recursion. The selection of this relational operator (between $<$ and $<=$) is likely to have no significant effect on run time behavior, yet has a tremendous impact on the quality of analysis performed during symbolic evaluation.

The two examples above provide some indication of the extreme sensitivity of the program analysis performed by symbolic evaluation to slight changes in the input program. The causes and solutions for the difficulties experienced differ for each example: In the first example, the difficulty is essentially that discussed in section 3.2.2: The symbolic evaluator has no means of maintaining the fact that successive values of *total* can only be associated at run time with a single value of *limit* (or, in the terminology of section 3.2.2, the symbolic evaluator is incapable of recognizing that the values of *total* and *limit* are "logically linked".) By relaxing the constraint that simulated control flow be merged as soon as possible after a control flow fork and by permitting independent paths of execution to continue throughout the *while* loop, the difficulty can be avoided. To avoid the extreme sensitivity exhibited in the second example, independent paths of execution could be used, or one could make use of "conditional contexts" in which information about program quantities is deduced from the fact that predicates are known to be true in the consequent of a conditional and false in the alternative.

It is clearly undesirable for symbolic evaluation to be highly sensitive to slight differences in the manner in which computations are phrased within the input program. It is hoped that additional empirical study of symbolic evaluation will reveal the extent and seriousness of this problem. If the issue proves sufficiently important, the creation of a more robust and consistent symbolic evaluation framework will become a priority in future modifications to PARTICLE.

5.5 Opportunities for Improved Analysis Strategies

5.5.1 Introduction

This section describes a few prospective methods for improving the quality of value analysis within PARTICLE.

5.5.2 Finer-Grained Loop Abstraction Strategies

Section 4.3.3.2.2.3 described the substantial degradation in the quality of program analysis that can result from the abstraction of iterative and recursive loops. During loop abstraction, program quantities changing during the iteration of the loop are repeatedly generalized until they reach a fixed point under loop iteration and generalization. Although PARTICLE currently allows some room for adjusting the precision of loop abstraction strategies by means of varying the height of the lattice associated with abstraction, there are means by which more flexibility could easily be added. In particular, loops frequently manipulate values over ranges of data (e.g., loop induction variables, array indices and pointers), and in many instances it would be desirable to support greater amounts of precision within the range subdomain of the Value domain: Currently, the range subdomain is associated with a height of just 1 – within all domains currently permitted by PARTICLE, the generalization of a range Value with any other Value not already approximated by the range Value yields a completely unknown Value ($\top$). Relaxing this condition to allow for range subdomains of greater (but still only finite) height could permit substantially more precision during loop analysis. In particular, program quantities could be repeatedly generalized until they adopt a range width closely approximating the range of values adopted by that program quantity within the loop at run time; within the current system, such values would be generalized to an initial range Value and would then be immediately promoted to $\top$ upon any subsequent generalization.

Raising the height of the range subdomain by itself does little to guarantee more precise loop analysis: If the elements of the range subdomain are expanded only incrementally with each iteration of the loop (as would typically be the case with loops in which variables are being successively incremented or decremented), aside from minor space advantages there is little to be gained by associating ranges instead of sets with program quantities. The fact that successive generalization of a program quantity during abstraction has led to the formation of a range, indicates that the set of possible values associated with that quantity within the loop is not capable of being expressed by the set subdomain. While it is possible that the set of possible values associated with that quantity at run time could be expressed by a range whose span is little larger than the span associated with the largest possible set approximation attempted during abstraction, it seems more likely that the quantity could be associated with a considerably larger set of possible values. Rather than gradually extending the range being manipulated by very small amounts, the extent of the range could be lengthened by much larger factors in an attempt to capture the set of values associated with a changing program quantity.

Although the span at which a fixed point is reached is unlikely to represent a *tight* bound upon the values associated with a quantity, it is likely to be a far more precise specification of that value than the $\top$ or *0-MAXINT* range Value that would be arrived at during conventional abstraction strategies.

Abstraction strategies groping for a range bound on a program quantity have considerable leeway in choosing the detailed path by which to successively generalize the ranges associated with program quantities. Simple strategies could successively enlarge a quantity's range by some fixed, user-specifiable factor until a fixed point is reached or the process has exceeded the legal lattice height by participating in too many generalizations. When coupled with range subdomains of larger height, smaller factors would permit more precise but slower abstraction, while larger factors would yield coarser estimates of the ranges associated with program quantities but support faster termination strategies.

More sophisticated range expansion strategies might attempt to make use of information available from the symbolic evaluation history of the program or from the program text to propose increasingly broad ranges which seem likely to encompass the value of the program quantity. Before simply searching blindly for appropriate ranges to encompass a program quantity by successive enlargement of the range being manipulated, such strategies would attempt to make use of any semantic information that might help bracket the appropriate range of values for that quantity. For example, an abstraction strategy might successively expand the ranges associated with program quantities to encompass values to which those quantities are compared within the loop (e.g., if a termination condition compares a program quantity against some known value outside of the range, that value would be proposed as an endpoint of the range associated with the program quantity) before beginning to simply beginning to double the span of each proposed bounding range for that program quantity.

5.5.3 Conditional Contexts

Previous sections have briefly mentioned an important means of acquiring knowledge about values associated with program quantities: When processing either the consequent or alternative of a conditional, it may be possible to exploit the fact that the truth condition of the predicate is exactly known. Assuming the truth or falsehood of a conditional's predicate when processing the consequent or alternative of that conditional. The ability to make such assumptions can play a surprisingly important role in sharpening compile time knowledge about the values of program quantities.

Figure 5-5 provided an example of a very simple case in which the knowledge available from conditional contexts would have prevented the need to abstract a recursive sequence of calls (see 5.4.) and the resulting loss of precision in analysis. As discussed in section 3.1.3.4, conditional contexts can also play an important role in eliminating repeated bounds and other domain checks. (See section 3.1.3.4.) Figure 5-6 illustrates an example of these benefits: Arrays X and Y are known to have the same size. A bounds check would normally be performed on index i upon each reference to each of the arrays; the ability to acquire knowledge through conditional contexts allows the symbolic evaluator to recognize that the success of the first bounds check logically implies the success of the later checks and thus permits the system to legally remove those subsequent checks.

Figure 5-8 illustrates a common scenario in which maintaining conditional contexts can

```
{
        int  X[1024]
        int  Y[1024]

        for (i = 0; i < k; i++)
           {
           X[i] = Y[i] * foo(X[i-1]);
           ...
           }
}
{
        int  X[1024]
        int  Y[1024]

        for (i = 0; i < k; i++)
           {
           assert(i < 1024);
           assert(i >= 0);
           assert(i < 1024);
           assert(i >= 0);
           assert(i-1 < 1024);
           assert(i-1 >= 0);
           X[i] = Y[i] * foo(X[i-1]);
           ...

           }
}
{
        int  X[1024]
        int  Y[1024]

        for (i = 0; i < k; i++)
           {
           assert(i < 1024);
           assert(i-1 >= 0);
           X[i] = Y[i] * foo(X[i-1]);
           ...
           }
}
```

Figure 5-6: The abstractions associated with array indexing within the code fragment above perform bounds checks prior to accessing the requested array element. The collection of information from conditional contexts provides a uniform and general framework for eliminating unnecessary bounds checks.

prove particularly useful during loop abstraction: When the symbolic evaluator has the ability to acquire information from conditional contexts, the system will recognize that i within the loop body is always less than 1000 and the abstraction of the loop will converge with i bound at the top of the loop to some fairly small range of values (ideally to the range 0-999, but such precision may not be possible within realistic loop abstraction strategies: See section 5.5.2.). In the absence of conditional contexts, the system has no means of recognizing the fact that i is always less than 1000 within the loop body and the system will model i as continuously changing at the top of the loop until the point at which the value of i at the top of the loop is abstracted to $\top$. The ability to use static information gained from conditional contexts thus has a first-order effect upon the precision with which i can be modeled following the abstraction of the loop. In general, it is frequently the case that the precision of loop abstraction will be considerably strengthened by the use of conditional contexts: Conditionals frequently exercise direct control over the termination of the loop. The maintainance of conditional contexts established by the loop termination condition will exclude program states matching the loop termination condition from reaching the top of the loop and will permit a stabilization of information at the top of the loop that approximates the actual set of possible states seen at that point at run time. In the absence of conditional contexts, however, no such stabilization is possible: The set of states associated with the top of the loop will continue to expand to embrace even those values of quantities known to result in the termination of the loop, and in most cases abstraction seems likely to continue until the quantities controlling loop iteration are abstracted to $\top$.

The ability to acquire knowledge from conditional predicates within PARTICLE would require moderate structural changes to the existing framework. Two major types of modifications would be needed: Changes to more completely separate the mechanisms needed to maintain and operate upon Values and knowledge concerning such values, and the construction of machinery to deduce knowledge about Values from predicate expressions.

Within the current system, knowledge concerning the possible values associated with a given run time quantity (“Value”, in the terminology of section 4.2.2.) is only acquired at the point at which the Value is created – there is no ability to sharpen the knowledge about a Value's run time values during subsequent processing. As a result, the knowledge concerning a given Value is same at all the points during symbolic evaluation at which it is in existence. Because a Value and knowledge about the values associated with that Value are coextensive within the current system, that knowledge can be maintained and manipulated as part of the Value itself. By contrast, the support of conditional contexts has as a fundamental prerequisite the ability to maintain different sets of knowledge about a given program Value within different symbolic contexts. (See figure 5-7.) Just as any symbolic evaluator supporting locations whose contents vary over different contexts must support some means of generalizing the contents of a given location at program join points, so any system supporting Values about which knowledge can vary in different contexts must support a means of generalizing the *knowledge about a particular Value* at program join points. (For example, if a particular value is known to be associated with the value 1 within the consequent of an *if* statement and with the value 2 within the alternative of that statement, that Value will be known to be associated with some value in the set { 1, 2 } in the code following the conditional.[11]) Only moderate effort is required to implement

[11]Note that there is a substantial semantic and practical difference between the ability to represent and maintain distinct sets of knowledge about the same Value and the ability to represent and maintain distinct

the bookkeeping machinery needed to separately maintain and operate upon Values and the knowledge associated with them.

```
{
        i = u;         /*  u unknown, associated with Value v */

        if (i > 0)        /*  Unknown which branch taken */
           {
           ...                 /*  Value v known to be in range 1-MAXINT */
           }
        else
           {
           ...                 /*  Value v known to be in range MININT-0 */
           }

        /*  Knowledge about Value v known is LUB(1-MAXINT, MININT-0) = TOP */
}
```

Figure 5-7: The ability to deduce contextual information concerning a Value requires the capacity for maintaining different knowledge concerning particular Values within different contexts and the potential for merging together such information at program join points.

In addition to changes allowing the segregation of information concerning Values and knowledge about Values, the support of conditional contexts requires mechanisms supporting the deduction of such knowledge from the predicates of conditionals. While the modifications necessary for separate maintainance of Values and knowledge about Values can make heavy use of machinery already existing within PARTICLE, the additions needed to deduce knowledge from predicate expressions require distinctly different mechanisms than are currently included within the system. Nonetheless, it is straightforward to construct a framework to deduce knowledge from simple predicates involving unary and conjunctive logical operators. The best means by which to extend such a system to handle general expressions (e.g., expressions involving OR operators) is unclear. One option is to make use of the ability of the PARTICLE desugaring mechanism (see section 3.5.4.) expand conditionals whose predicates contain non-conjunctive logical operators into sequences or nested series of simpler conditionals (at the cost of potential code expansion for each such construct). Within each of such simplified conditionals, precise truth conditions for values will be known for the particular subexpressions of the predicates.

knowledge about the distinct Values possibly associated with a given program resource (e.g., memory location). The distinction between these two capacities directly derives from the more basic distinction between a particular Value and knowledge about that Value that was discussed in section 4.2.2. On a more operational level, different instances of a single Value that happen to be associated with different knowledge concerning the possible values associated with that Value will always be associated with the same identity information (see section 4.2.2.1, while different Values will typically be associated with different identities.)

For conditionals in which the code expansion associated with desugaring is judged too severe, the original predicate could be used.

Supporting the acquisition of knowledge from conditional contexts requires not only the deduction of knowledge concerning program Values from the predicate expression, but also the ability to merge knowledge deduced from the predicates of a conditional with *existing* knowledge concerning program Values in order to allow conditional contexts to sharpen (rather than simply replacing) earlier knowledge. In a lattice-theoretic sense, the operations upon Value knowledge must be expanded to permit finding not only the LUB of the knowledge concerning two values (expressing the *generalization* of those two Values), but also the GLB (Greatest Lower Bound, or $\sqcap$) of those Values (expressing the *sharpening* of the Values). While adding support for the $\sqcap$ operation will require some effort, taking the $\sqcap$ of two Values is conceptually straightforward and the implementation should pose no particular difficulties.

The discussion above has focused on changes to PARTICLE that are logically necessitated by the support of conditional contexts. There is an additional set of changes to the PARTICLE analysis system that are not essential to the support of conditional contexts but may prove crucial for gaining substantial benefit from such a mechanism. In particular, any system making use of conditional contexts will only be capable of capturing knowledge from conditional contexts if the abstract Value domains employed by the symbolic evaluation system are capable of expressing the truth or falsehood of the predicates associated with the conditionals and if the symbolic evaluator can exploit such knowledge during the processing of the branches of the conditionals. For example, if the system encounters a conditional whose predicate tests whether a certain program pointer quantity is *not* equal to NULL, the system needs to be able to express and exploit this knowledge about the Value associated with the program quantity within the consequent for the conditional – a requirement that might require the augmentation of PARTICLE's Value domains by a "Not" Value subdomain. (See section 5.5.6.) Similarly, the ability to take advantage of a conditional test on the equality of two pointer variables would require that the system can exploit direct knowledge that two pointers are aliased or non-aliased regardless of other knowledge about the referents of those two pointers. The desire to consistently extract and exploit knowledge from common forms predicates may lead to the need to rework and extend some of the PARTICLE domains and knowledge representation mechanisms.

Conditional contexts offer the opportunity to substantially improve the quality of program analysis in certain important classes of situations, and provide a natural basis on which to support a powerful generalization of program specialization that is discussed in section 5.6.3.2.3. At the same time, the implementation cost for acquiring this powerful new source of static information is relatively modest. It seems likely that conditional contexts will be adopted as a source of knowledge about program values within future versions of PARTICLE.

5.5.4 "Backwards" Propagation of Static Knowledge

As discussed in section 5.5.3, future versions of PARTICLE may permit the acquisition of knowledge concerning program values not only at the points at which those values are created, but also at other points during the lifetime of those quantities. All extensions discussed earlier involved the acquisition of information about program quantities at points "down the execution stream" from an appropriate construct: e.g., A predicate within a conditional permitted the

```
...

for (i=0; i < 1000; i++)
        {
        ...                 /* Does not modify i */
        }
...
```

Figure 5-8: In this case, value identity records the fact that x and y are equal despite the fact that their exact values are unknown.

sharpening of static knowledge in points within and possibly after that conditional, and an assignment statement created knowledge about the quantity targeted by the assignment. In certain cases, however, a construct within the program may logically constrain the values of program quantities both *before and after* the position of that construct in the execution stream. In particular, the ability to make assumptions about program correctness may permit realization of constraints on possible values of a given program value throughout the lifetime of that value. Because the analysis methodology employed within PARTICLE is based fundamentally upon a form of "forward" static analysis, the system is unable to acquire information concerning program quantities from constructs which have not yet been reached within the path of execution. Figure 5-9 shows an example where semantic constraints logically permit knowledge of program values both before and after the execution of the constraining construct. Unfortunately, PARTICLE's bias towards forwards flow analysis prevents the system from easily recognizing and exploiting such information.

In addition to preventing the recognition of constraints on the values associated with program quantities for the full region of execution in which those constraints apply, PARTICLE's inability to propagate backwards static analysis information complicates the implementation of important optimizations such as dead code elimination that are dependent upon information gathered during a backwards flow analysis. Although PARTICLE currently relies upon the postprocessing compiler system to perform optimizations requiring such analysis, it would be desirable to investigate the possibility of incorporating such optimizations into a more powerful analysis framework such as that offered by PARTICLE. It remains unclear whether the analysis necessary for such optimizations can be implemented in as precise and full-bodied a manner as the analysis performed for optimizations relying only upon information gained through forward flow analysis. In particular, fine-grained knowledge concerning program behavior (e.g., the number of iterations executed by a given loop, or the direction taken by a program branch) would appear to rely intimately upon information only available during *forward* static analysis; it not clear how backwards static analysis seeking to take advantage of precise knowledge of program behavior could proceed in the absence of large amounts of forward flow information. On the other hand, it may be that that high quality forward static analysis such as that performed by PARTICLE could allow subsequent backwards flow analysis to take advantage of

```
{
        int array[1024];
        int i;

        i = u;          /*  u unknown */

        ...             /*  within this code the system should have access to
                            information gained about i from the semantic constraint
                            below */


        a[i] = foo;     /* semantic constraints => 0 <= i < 1024 */

}
```

Figure 5-9: The program fragment shown illustrates a case in which the capacity for propagating static information backwards within a program can help to sharpen knowledge about program values. The assumption that the input program is semantically correct would legally allow the system to recognize that $0 \leq \mathrm{i} < 1024$ before the point at which the array assignment actually takes place.

the information gained about program behavior, and permit the possibility of augmenting and refining that information with additional knowledge gained through backwards analysis.

There are many ways in which backwards flow analysis algorithms might potentially be integrated into the current PARTICLE framework – as back-annotations performed during forward flow analysis (see section 5.6.2.2.), as part of a separate backwards flow analysis pass making use of high quality flow information gathered through earlier forward passes, or as a backwards pass operating in a more traditional framework and without access to the intimate knowledge of program behavior gained through earlier analysis. The problem of discovering appropriate frameworks for different flow analysis problems constitutes a challenging open problem for future research.

5.5.5 Improved Memory Model Representations

The fundamental structure of the memory model currently employed by PARTICLE was discussed in section 4.2.3. Although the current scheme is adequate for capturing knowledge concerning program state, it makes uniform use of very fine and expensive representation even in those cases where such careful modeling of the store offer little or no benefit. Moreover, the current memory model is incapable of capturing certain information about program quantities that is commonly available to traditional compilers. Each of these shortcomings are briefly discussed below. For the sake of brevity, the discussion avoids discussion of second-order extensions to the current system (that is, extensions that serve only to improve other potential additions to PARTICLE.)

5.5.5.0.1 Overgenerosity in Representation The existing memory model attempts to model the layout of the store as precisely as possible, adhering in most instances to a representation that simulates that store on a location-by-location basis[12]. Although this representation guarantees the availability of fine-grained knowledge about memory contents wherever it is needed, it can waste considerable compile time resources in situations where the available static knowledge is insufficient to make use of the advantages offered by such a fine-grained specification of program state. In such situations, a far more compact representation would be capable of capturing as much information as the existing representation at a small fraction of the space usage.

As an example, consider a large array whose individual elements are associated with completely unknown values: The contents of the array could be represented in a highly compact manner, and the representation could be adaptively modified to capture information about the contents of individual elements as such knowledge was obtained. In other cases, it may be desirable to create special representations for arrays whose elements are sparsely known, or arrays whose values possess uniform values. Each of these array representations can be naturally extended to permit compact modeling of general-purpose contiguous regions of memory.

As an alternative or complement to techniques representing uniform or sparse regions of memory via ad-hoc mechanisms designed to express such conditions, it may be desirable to consider compact expression of information about regions of memory deemed as less interesting by "folding locations on top of each other" so that the contents of many memory locations are LUBed together and stored as a single value representing a legal approximation to the contents of each of those locations. While such an approximation could lose a great deal of information in certain ranges of circumstances, in other cases it may permit simple approximate representations of data which is otherwise difficult to express compactly. For example, while it may not be desirable to exactly represent a complex multi-piece pointer data structure, maintaining a pseudo-component representing a LUB of all of the actual component would permit capturing certain important characteristics about the possible referents of the pointers within that data structure. (For instance, such a representation could tightly bound the possible effects of an indirect through one of the pointers in the structure and could capture the fact that all pointers within the structure pointed within the structure itself rather than to external regions of memory.) As another example, consider an array each of whose items is associated with some possible set of small integral values, but in which there is considerable difference between the values associated with each element. It seems likely that a LUB representation of all of the values within the array would lose little important information about program behavior, while still capturing some important properties of that data (e.g., that all of the values represent legal indices into some other array).

5.5.5.0.2 Capturing Information About Referents of Inexactly Known Pointers The section above discussed the considerable amounts of unneeded space overhead that can be

[12]Exceptions to this tendency arise within the representation of state during the abstraction of recursive function calls, as discussed in section 4.3.5.2.

associated with the current memory model, and discussed means for lessening that overhead with little or no cost to the quality of analysis performed during symbolic evaluation. This section turns to examine the possibility of considerably *improving* the fidelity of the current memory model in certain common situations. Consider the code fragment in figure 5-10. Although the referent of pointer p may not be known, it is straightforward for a human onlooker to statically recognize that after the initial write, the value v is associated with all occurrences of the rvalue $*p$. Unfortunately, PARTICLE is not currently capable of recognizing this invariant: The initial indirect write through p will be treated as a write to any one of a number of possible locations, and each of these location will be associated with the LUB of its current contents and the written value. All subsequent reads from the pointer p are treated as reads from a number of different possible locations, yielding a value that is the LUB of the values in those locations. Thus, unless *all* of the values in the set of possible referents were *originally* associated with the value 0, the value indirectly read through pointer p will not itself be 0 but will instead by some inexact approximation to 0 that simultaneously approximates the values initially associated with each of the possible referents of p. What is lost here is the recognition that the write through p occurs to *some particular memory location*, and that the subsequent reads through p are reads from *the same particular memory location that was just written with the value 0.* This is a particularly common and exasperating illustration of the shortcomings discussed in section 3.2.2: Unfortunately, there is no confluence between the LUB of the memory models resulting from the separate simulation of the pointer operations on each of the possible pointer values, and the memory model resulting from the simulation of the pointer operations on a pointer representing the LUB of the possible pointer values.

Although section 3.2.2 discussed the potential for sharpening program analysis and lessening impact of similar situations by allowing for analysis in which the LUBs of program states are performed considerably later than in the current implementation of PARTICLE, such a system would be associated with considerably greater compile time overhead and would merely lessen the severity of but not eliminate the difficulty discussed above. Because of the importance of pointer-based code within C programs, it seems desirable to find a more complete solution to this particular problem, even if that solution does not addresses the more general confluence issues. One possible means of addressing the difficulties above involves the association of a "pseudo location" with each inexactly known pointer Value. This pseudo location would record as accurately as possible the contents of the particular location referred to by its associated pointer, *even if the exact address of that location is not known.* The handling of read and write requests to the pseudo location is relatively straightforward and is reminiscent of the handling of analogous requests on systems with multiple caches associated with possibly overlapping sets of cached locations. In particular, when a read to an inexactly known pointer occurs, the system can simply return the value in the associated pseudo-location: Although the system does not know which particular memory location is associated with the read pointer, it has exact knowledge of the contents of that pointer. The process of writing through an inexactly known pointer is somewhat more involved due to the fact that the sets of locations possibly associated with different pointer Values may not be disjoint: It may be that the memory locations adumbrated by two distinct pseudo locations may overlap. Because of this possible "aliasing" between pseudo locations, it is necessary to conservatively record the effect of a pointer write on all possibly affected pseudo locations. Thus, whenever a write occurs to a particular inexactly known pointer, the contents of the "pseudo location" are overwritten with that value, all of the

```
{
        int array[1024];
        int *p;
        int i;

        /*  p not fully known */

        *p = 0;

        for (i = 0; i < 1024; i++)
           {
           array[i] = *p;     /*  system cannot recognize that *p has value of 0 */
           }
}
```

Figure 5-10: The fragment of code involves a write to some pointer associated with an unknown referent, and a series of subsequent reads from that same pointer. Traditional analysis systems would be capable of recognizing the fact that all reads through pointer p would yield the quantity written to that pointer at the top of the fragment. Unfortunately, the current version of PARTICLE is unable to recognize this invariant: The symbolic evaluator has no means of definitively recording the value pointed to by a pointer in the absence of exact knowledge concerning the exact identity of the location referred to by that pointer.

possible real locations possibly associated with the pointer have their contents LUBed with the value being written, *and* all pseudo locations which are also possibly associated with any of the real locations written to during the write (that is, any pseudo locations associated with pointers possibly sharing referents with the current pointer) have their own contents LUBed with the value being written. Although the implementation of the "pseudo-location" mechanism requires some bookkeeping machinery, the method simultaneously maintains two important sets of information: High-quality modeling of the contents of the locations in the memory model, and the ability to capture knowledge about the contents of the locations referenced by pointers even if the identity of those locations is inexactly known.

5.5.5.0.3 Conclusion The adoption of the techniques outlined above offers the potential for simultaneously reducing the space required to represent program state during symbolic evaluation and for increasing the fidelity of that representation. In view of the considerable space demands associated with symbolic evaluation under PARTICLE and the importance of modeling pointers in a high-fidelity manner, it seems likely that both of these additions will be adopted in future version of PARTICLE.

5.5.6 Richer Value Representation Strategies

This section briefly surveys a variety of possible extensions to the PARTICLE abstract domains offering the potential for improving program analysis in a variety of different situations. Space considerations prevent all but the must cursory discussion of each possible extension, and rule out the mention of all but the most promising candidates.

- ***Not* Representations** A potentially valuable abstract Value subdomain not currently available within PARTICLE would permit the system to capture the fact that a Value is known *not* to be associated with a specified value (bit pattern). The "not" subdomain would lie just below the $\top$ element in the Value domain and was present in at least one previous partial evaluation system [25]. The ability to represent non-equality between a Value and a particular known value permits capturing of conditional context information in those common cases in which the conditional is associated with a predicate testing the equality (or non-equality) of a Value with some particular value. (See section 5.5.3.) It is possible that this representation (or some variant on it) will prove particularly important for representing pointers, by allowing the system to capture the fact that two pointers are known not to be aliased or that a given pointer value is not equal to $NULL$.

- ***Bitset* Representations** Within programs that make extensive use of bitwise operations, it may be desirable to support the use of a subdomain which maintains distinct knowledge about particular bits within program Values. For example, it may be that a particular program quantity is known to have all of its less significant bits set to zero, while the most significant bits are associated with unknown values. A representation that permitted compact expression of such knowledge could allow faster and more space efficient processing of bit-intensive code than would be possible with set-based representations of the same information. In particular, while the cardinality of a set representation of bitwise knowledge would increase by a factor of 2 for each unknown bit within the Value, a bitwise representation could represent such knowledge within space proportional to the number of bits associated with the Value. Of course, the bitset representation achieves such compactness at the expense of generality: While bitsets can concisely characterize Values associated with knowledge cleaved along bit boundaries, they serve as a poor substitute for Set representations when describing arbitrary collections of values.

- **Modulo Representations** A further addition to the domains of the current system could be based upon a subdomain representing the fact that a Value is known to be associated with bit representations that are *n modulo m* for some specified m and n. For instance, the expression $4 * v$ is known to produce Values that are *0 modulo 4* for all integral v. Figure 5-11 illustrates a case in which access to a modulo representation of data can considerably decrease the amount of static information lost as a result of a pointer write.

  Although a simple modulo representation can capture considerable knowledge about the values associated with a program quantity, a generalization of this idea is likely to prove considerably more desirable. In particular, while it may occasionally be desirable to characterize a program value only by virtue of its modulo value, considerable extra flexibility and expressiveness would result from combining the modulo representation with the integral range representation, and permitting the modulo information to be augmented with some lower and upper bounds on the magnitude of the Value. It seems plausible that many values currently represented inefficiently as general sets of values could be compactly expressed within the new subdomain. For example, consider the effects of multiplying an integer known to be within some range of values by a constant, or of representing the pointers to the elements of some array whose elements of are of some fixed size greater than 1 – each of the resulting values could be concisely represented within a modulo-based range representation, while the accurate representation of such values within a set representation would consume considerably more space. This generalization of the range representation for integers seem likely to prove sufficiently beneficial to repay the slight changes involved in augmenting the current implementation.

```
{
        double array[k];

        /*  i unknown */

        array[2 * i] = 0.0;          /*  this invalidates the elements at */
                                     /* even slots within array */
}
```

Figure 5-11: Modulo range representations provide a natural framework for describing the results of multiplying an integer by some constant or a range of pointers to aligned data.

- **Symbolic Representation and Reasoning about Knowledge**

  There are several possible extensions to the current system that rely upon the existence of mechanisms permitting reasoning about general relations between incompletely known Values based not on knowledge concerning the particular values (bit patterns) thought to be associated with those Values, but rather on more abstract or symbolic knowledge of relations known to hold between the Values. Rather than separately capturing knowledge for each program quantity and combining such information wherever it is needed, a system maintaining symbolic information will have the capacity for manipulating explicit predicates involving such quantities or their associated Values. Each of these extensions is briefly detailed below:

  – **Simple Symbolic Reasoning about Unknown Program Values** Consider the code fragment shown in figure 5-12: In this example, it would be clear to any human observer that under all (non-overflow) conditions the loop will be iterated 8 times, no matter what the actual value of k involved in the loop control. Unfortunately, the form of symbolic evaluation used within PARTICLE is not be capable of recognizing this invariant: Upon entry to the loop, the expressions k and $k + 8$ both evaluate to unknown Values, but the system is incapable of reasoning about the relative magnitudes of and differences between such unknown quantities (e.g., the system will fail to recognize the fact that $k <= k + 8$, and would be incapable of realizing that adding 1 to the initial value of k eight times will yield the value $k + 8$.)

```
/*  k unknown */

j = 0;
for (i = k; i < k + 8; i++)
   {
     array[i] = Foo(j++);
    }
```

Figure 5-12: Because the value of k is unknown within the code fragment above, PARTICLE is incapable of recognizing that the loop can be legally unrolled 8 times and specialized accordingly. The addition of machinery to allow the symbolic reasoning within program analysis would permit the system to recognize such invariants.

A variation of the same problem is shown in figure 5-13. In this example, the predicate $i == k$ will always be true, and the conditional can be safely collapsed. Unfortunately, because PARTICLE has no general means of detecting or representing the relations between inexactly known values, this invariant will not be recognized by the system. Ideally, during abstraction of the *for* loop, the variable i would be successively generalized to larger and larger sets of possible values until an extra simulated iteration of the loop no longer changes the Value of i within the loop. The maximally precise point at which this could happen is when the Value of i at the top of the loop is known to be possibly associated with any value in the range 0 to the Value of k. At this point, the value of i *inside* the loop will be associated with the range 0 to $k - 1$ (because a Value of $i == k$ would lead to loop termination)[13]; generalization after the iteration would yield an identical loop entrance approximation to i as was present upon the previous iteration and would mark the convergence of the evolution of i to a fixed point. Unfortunately, because of the limitations on explicitly representing and manipulating relations between incompletely known Values within the current system, it is at no point possible to generalize the representation of the Value of i at the top of the loop to yield the knowledge that i varies from 0 to k. As a result, PARTICLE will generalize the Value of i until the point where it reaches the Value $\top$ (or the range 0-MAXINT)

[13]Note that this reasoning makes use of the assumption that mechanisms supporting the existence of conditional contexts are in place: See section 5.5.3.

```
for (i = 0; i < k; i++)                /* k > 0, otherwise unknown */
        ;

.
.
.


if (i == k)                /*  always true */
        {
        ...
        }
```

Figure 5-13: Although PARTICLE can reason about relations between exactly known program Values, it is incapable of capturing knowledge concerning relations between inexactly known Values other than identity. Within the example above, abstraction of the loop is carried out only upon elements of PARTICLE's abstract domains and the system will fail to recognize that $i == k$ following the abstraction of the loop. Such limitations hinder PARTICLE from discovering many "common sense" opportunities for specialization.

Abstractly viewed, the problems discussed above can be attributed to the lack of confluence discussed in section 3.2.2: In the two examples above, the specializations discovered by simulation of the code in which k is simultaneously associated with the LUB of its possible values are far less aggressive than those that would be discovered were separate paths of analysis to be followed for each of the possible values of k. However, while the shortcomings illustrated above represent merely one manifestation of a general phenomenon, the serious character of these difficulties may warrant a special-purpose investigation into pragmatic and ad-hoc mechanisms for decreasing their cost (in the same spirit as motivated the suggestions for improved memory modeling discussed in section 5.5.5.0.2.). While any method for addressing such shortcomings will potentially offer significantly improved reasoning about program values in common situations, such a technique will also have confront the fact that maintaining and reasoning on the basis of symbolic information such as that described above may ignore issues such as overflow and round-off error which can potentially lead to divergence in behavior between the computations performed at run time and the symbolic simulation of those computations.

– **Composition of Symbolic Knowledge** The code in figure 5-14 illustrates another aspect of symbolic reasoning concerning program Values: Once the system is capable of expressing knowledge concerning symbolic relations between Values, it may be desirable to allow for deduction of new knowledge concerning those Values from preexisting knowledge. In the example shown, even if PARTICLE were extended to allow expression of the relation between two incompletely known values, it would not be able to take full advantage of the static knowledge available from the program text: In order to limit the effects of the array write, the mechanisms within the symbolic evaluator would be required to recognize and exploit the transitivity properties of relational operators. The benefits stemming from the ability to symbolically express knowledge about the relations between program quantities could potentially be substantially enhanced by supporting the deduction of further knowledge from such relations.

```
/*   i, j, k unknown */
{
        int   array[k+20];

        if (i < j)
                if (j < k)
                        {
                        array[i] = 0; /*  only affects PART of array */
                        }
        ...
}
```

Figure 5-14: The application of symbolic reasoning within the analysis of the code fragment above permits the system to recognize that the effects of the array write are bounded to a subset of the array.

– **Representing Knowledge About Program Locations**

Within this general discussion of the benefits offered by the symbolic representations of knowledge, it may be instructive to briefly consider the advantages offered by representing symbolic knowledge concerning program locations and general expressions rather than about the Values within those locations or produced by those expressions. The primary benefit of such a means of expressing knowledge lies in its ability to avoid some of the problems with confluence discussed in section 3.2.2: By capturing knowledge in terms of *locations* rather than values, it is possible to express relations and conditionals that abstract away from the particular Values associated with those locations and are less subject to disruption by the generalization associated with join points within the program. Consider figure 5-15: Within the code fragment shown, the current system (or any of the extensions to that system discussed above) will fail to permit constant folding of the expression $a + b$. (See section 3.2.2.). While it is possible to address this difficulty by pushing the meet point

```
if (flag)       /* predicate unknown */
           {
   ...
   a = 1;
   b = 3;       /* a+b == 4 */
   ...
   }
else
   {
   ...
   a = 2;
   b = 2;       /* a+b == 4 */
   ...
   }

/*  after join point

a = element of {1,2}
b = element of {2,3}
*/

/*  c will ALWAYS be 4, but PARTICLE will
           not recognize this.
        */

c = a + b;
...
```

Figure 5-15: Within the code fragment above, PARTICLE will be unable to recognize that $a + b$ always equals 4 – at the meet point, a and b will each be generalized to be known as an element of a set of two possible values, and PARTICLE will perceive $a + b$ as yielding one of three possible values.

for the symbolic evaluation streams further into the code following the conditional, such an option increases compile time overhead and serves only to lessen but not eliminate the problem. A strategy that augmented the current mechanism by also permitting the capturing of knowledge concerning the values associated with symbolically specified expression involving names for program quantities would avoid this difficulty. Within such systems, knowledge that $a+b == 4$ would be associated with the symbolic evaluation of each side of the conditional, and can therefore be legally maintained after the meet point – even if the *particular values of a and b involved were not fully known.* Because they abstract over the particular Values associated with each program quantity, such location-oriented predicate descriptions tend to be insensitive to the loss of information about Value simultaneity that occurs at meet points during symbolic evaluation. (See sections 4.3.3.2.2.3 and 3.2.2.) [14]

Although the symbolic representation of knowledge about program locations permits would permit the symbolic evaluation system to avoid many of the confluence difficulties associated with knowledge centered upon Values, the formulation of the methods by which these symbolic representations are created and maintained will require significant thought.

At any given point during symbolic evaluation, there are typically a large variety of symbolic predicates which could be deduced, only some small fraction of which would actually prove useful during subsequent symbolic evaluation. For example, in order for the code fragment within figure 5-15 to be properly specialized in the manner described above, the system must be able to deduce that the predicate $a + b == 4$ is true for both of the branches of the conditional. It would be strongly preferable if such deductions could take place without the need to deduce a myriad of other unneeded predicates (e.g., the value of $a - b$, $a * b$, and a/b for each branch). It is not obvious exactly what heuristics can or should be used to focus the deduction of symbolic location-oriented predicates in such a manner that only those predicates which seem most likely to offer the potential for specialization will be deduced. One plausible constraint upon predicate creation would restrict the system to considering only the deduction of those predicates involving “live expressions” (that is, expressions which will be encountered during subsequent control flow without having any of the program quantities involved changed by variable writes). This constraint could enormously reduce the number of predicates deduced within the system while retaining those predicates which seem to offer any possibility of permitting subsequent specializations. Unfortunately, determining what expressions are live requires backwards

[14]Note that the fundamental advantages offered by the ability to maintain predicates explicitly involving program locations are conceptually closely related to the advantages associated with the maintainance of pseudo locations (see section 5.5.5.0.2.): In both cases, ad-hoc mechanisms are used to make the knowledge of the symbolic evaluator insensitive to loss of simultaneity information, and the underlying means by which this desensitization is accomplished are similar: In the case of pseudo-locations, the system maintains information concerning the referent of an inexactly known pointer Value that abstracts across all particular *values* that Value may take on. When representing knowledge in the form of explicit predicates concerning program locations, information about relations between program quantities is maintained in such a manner that it abstracts across the particular all particular *Values* (*and* values) associated with those quantities. The ability to abstract across a certain level associated with a particular object voids the need to worry about the loss of simultaneity information at that level.

flow analysis information, and would thus not admit straightforward integration into the current system. (See section 5.5.4.) Nonetheless, if the implementation of such a heuristic is feasible, it would appear to offer significant promise for allowing access to significantly greater amounts of static information while restricting such knowledge to those pieces of information that are most likely to be useful in program specialization.

The ability to correctly maintain symbolic information concerning program locations requires significant bookkeeping machinery. The LUB of all predicates must be taken at program meet points, and predicates may be invalidated or modified when side effects change (or *may* change) the contents of the locations to which they refer. Although it is straightforward to construct a simple implementation of these mechanisms, more sophisticated schemes capture additional information concerning program quantities but require substantial additional reasoning. For example, the LUB of two predicate databases concerning a given location may be simplistically implemented as the intersection of those databases (because it is always safer to err on the side of knowing *less* about a program quantity). Richer schemes may represent the LUB of two different predicates as a distinct third predicate. (e.g., the LUB of $a+b == 4$ and $a+b == 5$ might be treated as $a+b > 3$). Creating and maintaining correct and rich predicate information can require substantial complexity in the supporting mechanisms.

This section has surveyed a wide variety of means for enhancing PARTICLE's capacity to represent knowledge about program quantities. These modifications are associated with a spectrum of implementation costs and expected payoffs; several of the changes need considerable additional exploration before their implementation can be considered, while others are conceptually straightforward. It is unlikely that PARTICLE will ever come to incorporate even a majority of these possible modifications, but the ideas behind them may suggest possible routes for the future evolution of the project.

5.6 Enhancing PARTICLE's Optimization Repertoire

5.6.1 introduction

the previous section discussed the potential for improving the quality of the static *analysis* performed by particle, with the hope that such analysis would permit the discovery of additional cases in which some element of a fixed set of optimizations will be applicable. this section turns to examine the possibility of enriching the set of optimizations performed by the symbolic evaluator. just as was the case in the discussion of improved analysis methodologies, the suggested changes span a wide range in terms of their ease of implementation, concreteness, and expected return and are far too numerous to consider in any depth. the following section briefly surveys a number of these proposed extensions to particle's capacities for optimization.

5.6.2 Incorporating Traditional Optimizations

PARTICLE currently delegates a wide range of traditional compiler optimizations to the post-processing C compiler. While this approach permits the program specializer to avoid duplicating the analysis and optimization machinery present in existing compiler systems, the postprocessing compiler has no access to the highly precise analysis information gathered during symbolic evaluation and may therefore be incapable of recognizing important opportunities for optimization. It is therefore natural to consider incorporating some of these external optimizations into PARTICLE itself. Not all optimizations fit cleanly into the symbolic evaluation framework – transformations such as code hoisting could be performed by PARTICLE, but would need to be somewhat shoehorned into the current framework. Nonetheless, PARTICLE can be extended simply and naturally to perform the analysis and transformations for several important classical optimizations, while other optimizations can be added with relatively minor changes. The opportunities for performing several of these optimizations are briefly described below.

5.6.2.1 Copy Propagation and Common Subexpression Elimination

Both copy propagation (CP) and common subexpression elimination (CSE) can be implemented in a relatively straightforward manner within PARTICLE by making use of the "identity" information associated with each program Value. (See section 4.2.2.1.) If the tag of one Value is equivalent to the tag of another Value, the run time values associated with each Value are guaranteed to be equivalent. Section 4.2.2.1 describes the mechanism in PARTICLE which guarantees a deterministic mapping of input Values to an output Value for a given expression: If a particular expression ever combines the same input Values and returns an output Value with a certain identity tag, all subsequent applications of that expression to those input Values are guaranteed to yield the same output Value (that is, a Value with the same identity tag). The mechanism guarantees that all common subexpressions (and, more broadly, all generalized common subexpressions – all expressions manipulating equivalent Values) will yield Values known to be equivalent. Thus, common subexpressions can be recognized by the fact that an expression evaluates to a Value recorded as having been returned by an earlier evaluation of some expression.

It should be emphasized that the class of common subexpressions discovered by PARTICLE represent generalizations of the class of common subexpressions uncovered by traditional compilers. In particular, the subexpressions potentially detected by PARTICLE might be characterized as "dynamic common subexpressions," and form a much broader class of subexpressions than the "textual" subexpressions discovered by most common subexpression analysis algorithms. This generalization has two major advantages: It allows PARTICLE to detect expressions returning identical Values even if the computations building up those Values differ greatly (one of the simplest results of which is the conceptual folding of copy propagation into generalized CSE), and it permits PARTICLE to recognize opportunities for code hoisting from loops by the same mechanism by which the system recognizes other subexpressions. Figure 5-17 provides two examples illustrating cases in which PARTICLE's more generalized notion of common subexpressions offers the system a uniform framework of recognizing opportunities for optimization that are not visible to traditional compiler CSE analysis.

Following the discovery of common expressions and copies of variables, the symbolic evaluator must annotate the program code with specializations needed to take advantage of the system's knowledge of the equivalence between Value. If a program quantity within the current state is associated with the return Value of the expression being recalculated, a reference to that program quantity can be directly substituted for the expression. In other cases, it may

```
c = b * a;        /*  a known == 1 */
e = c + d;

...

f = b + d;        /*  PARTICLE detects CSE here:  b+d already
                      calculated above.
                  */
```

Figure 5-16: The example above computes b + d in two different manners. While schemes relying on textual comparisons of expressions within the program text would be unable to recognize a common subexpression, the recomputation would be immediately obvious within PARTICLE's identity-based CSE detection mechanism. PARTICLE's approach to detecting generalized CSEs also has the added benefit of automatically allowing the detection of opportunities for copy propagation.

```
while (.....)
        {
        a = b * c;
        ...              /*  b, c not recomputed */
        }
```

Figure 5-17: PARTICLE's ability to detect "dynamic common subexpressions" permits the system to recognize that a given expression within the program code is being evaluated to the same value at each point of execution. The invariance of the expression Value may suggest an opportunity for code hoisting.

be necessary for the specialization to introduce a temporary variable to carry the Value from the last point at which it was available to the current point within the code. In some cases, the introduction of such a Value "transport mechanism" will be very simple, requiring little more than the insertion of a declaration and subsequent assignment statement. In other cases, transporting the Value may be far more involved (e.g., if the original expression evaluation was deep inside some series of function calls which have now returned). Some heuristics will clearly be needed to allow for specializations permitting effective common subexpression elimination without the need to develop elaborate strategies for handling complicated transport mechanisms. A simple but satisfactory strategy might simply rule out the possibility of transporting the Values associated with common subexpressions across procedure boundaries.

5.6.2.2 Dead Code Elimination

PARTICLE currently makes no attempt to avoid the output of dead residual code, and large amounts of dead code are routinely produced by the system. Unfortunately, the quality of dead code elimination in even highly optimizing compilers is not always as high as one would like – while PARTICLE may be capable of recognizing a particular pointer write, global memory access, or loop as unnecessary, the quality of static analysis within traditional compilers may be insufficient to provide the compiler with a guarantee that it is safe to eliminate the associated construct. It is therefore natural to consider performing dead code elimination within PARTICLE itself in order to permit it to take advantage of the high quality static analysis performed during symbolic evaluation.

Classical dead code elimination relies upon backwards flow analysis in order to recognize live variables and elide assignments to those that are not. As discussed in section 5.5.4, it remains an open question as to how to perform general backwards flow analysis within frameworks similar to those used within PARTICLE. Nonetheless, it is possible to perform certain common patterns of backwards analysis within a symbolic evaluator in a straightforward manner, and the analysis necessary to perform dead code elimination can be integrated into PARTICLE in a very clean and simple manner.

Briefly, the analysis for dead code elimination can take place as follows: Each statement maintains two sets for each call to its enclosing function. The first set describes the set of all locations possibly affected by the statement during that call; the other set specifies all locations possibly referenced by the statement during the call. (Note that this definition generalizes to apply to structured control flow and compound statements in a straightforward manner.) As statements are executed, each location can conceptually be associated with the identity of all of the statements that merely *possibly* wrote to that location back in the execution stream until the point where a *definite* write to that location occurred, and is also annotated also with the identity of the most "recent" (in a control flow sense – there may be many) statements that *definitely* wrote that that location. Whenever a statement is executed, all statements annotated to each of the locations being *read* is marked as "needed *for the call being evaluated*". For all all weak write side effects of the statement (see section 4.2.2.5.2.), the identity of the current statement is simply added to the lists of statements associated with each memory location being possibly weakly written. For all of the strong write side effects of the statement, the list of all existing statement identity annotations is erased for each of the target locations, and the current statement is recorded as the sole writing statement for those locations. At program confluence points, the statement sets associated with each location in the two memory models being LUBed must be unioned together to form a single statement set for LUBed location.

When symbolic evaluation terminates (but before the specialization decisions are made), each statement in the program is examined. The "needed" marker is checked for each call to the function enclosing that statement. For each call for which the "needed" marker is not set (indicating that the statement was *definitely* not needed for the entire call), the statement is annotated with a specialization that proposes eliminating the statement for that call by reducing the statement to the null statement. Following the examination of all statements within the program, control is then transferred to the specialization phase.

5.6.2.3 Inlining of Function Calls

Within traditional compilers, function calls are associated with a double penalty: In addition to the "raw overhead" associated with the parameter passing and transfer of control to the called function, a function call typically serves as a major barrier to effective static analysis. In such compilation systems, the inlining of function calls can therefore allow for markedly improved static analysis and optimization within the called function and can lead to dramatic performance enhancements.

As discussed in section 4.3.5, during symbolic evaluation PARTICLE transparently crosses function call boundaries. Because the quality of static analysis within PARTICLE performed by the system is unaffected by the presence of a function call, it is to be expected that the advantages offered by inlining code processed by PARTICLE will be much smaller than those associated with inlining in traditional systems. Moreover, inlining is an optimization easily and efficiently performed by existing compilers; unlike the optimizations discussed above, inlining would tend to benefit rather little (if at all) from the additional static information available within the symbolic evaluator.

Nonetheless, it would be straightforward to integrate inlining into PARTICLE's specialization framework, and there are several minor but not inconsequential motivations for considering such an option:

- Because PARTICLE is aware of which functions were in the original program and which have been created as specializations of those original functions, it has the option of entirely eliminating the maintainance of separate bodies for specialized functions all of whose call sites are inlined: By contrast, a postprocessing compiler is required to assume that any function could be called from outside the program, and must preserve the body to the function even if all calls to that function within the program are inlined. Performing the inlining of functions within the program specializer thus offers the ability to avoid the space overhead of representing each specialized but inlined function within the residual program.

- Making inlining decisions within PARTICLE would permit such decisions to be made within the cost/benefit framework explicitly trading off space usage and time savings.

- While selecting functions to inline, PARTICLE could take advantage of the knowledge that the function being inlined is unlikely to be able to be additionally specialized following its integration into the calling function; traditional compilers might mistakenly anticipate substantial optimizations following the inlining.

A simple strategy for making inlining decisions would make use of the fact that the costs and benefits of inlining are not likely to differ dramatically from the costs and benefits of specializing the called function to its calling site. The strategy would make all specialization decisions as normal – that is, as if all specializations were to be realized in the form of distinct functions. Following the system's convergence to particular specializations but preceding the emitting of the residual code, an inlining postpass would be run. This postpass would use heuristics to completely inline all calls to functions with few call sites, and would preferentially inline other specializations invoked by call sites associated with a large number of estimated run time calls in order to decrease the absolute performance cost associated with those call sites. Although it fails to take explicit account of the estimated time savings associated unique to inlining, such an inlining postpass would capture some of the greatest advantages of inlining and can be added in a modular and straightforward fashion to the current system.

5.6.3 Support for Speculative Specializations

5.6.3.1 Overview

This section considers a generalization of PARTICLE's optimization strategy that permits the exploitation of a substantially broader class of specializations than are permitted within the current system and opens up the potential for dramatic additional performance enhancements. The fundamental idea behind the generalization of the specialization strategy is simple: PARTICLE currently collects information on the possible values of program quantities, and makes note of instances in which the system's knowledge about program quantities reveals that an optimization is (definitely) valid. It is possible to envision a system that extends this process by modifying the specializer in such a manner that it not only recognizes and exploits cases in which specializations are known to be possible given its information about program quantities, but where the specializer also seeks to discover cases where *if* it were to know some additional information about the program's values, important specializations could be performed. In particular, such a system could recognize that while its own (compile time) knowledge about program state was inadequate to guarantee that a certain optimization or set of optimizations were possible, information about the legality of proposed specializations will *always* be available at run time. In certain cases, the expected performance enhancements associated with a of the specialization may be sufficient to make it beneficial to suffer the overhead of performing checks on the legality of that specialization at run time. In particular, the system could insert *run time checks* to see if the conditions necessary to perform a given specialization obtain at run time. If those conditions *are* detected to obtain at run time, the system can use the appropriately specialized code; in cases where the inserted conditional detects that the conditions for legal specialization do not hold at run time, the system must make use of code that does not take advantage of those specializations.

To a first degree of approximation, the current version of PARTICLE can be seen as attempting to shift the maximum possible amount of computation from run time to compile time: Wherever compile time checks indicate that the information accumulated by the symbolic evaluator is sufficient to permit pieces of the the program to be executed, the computations are performed at compile time and a residual program specialized to the known data is created. A version of the system generalized in the manner described above would permit the validity checks for optimizations to themselves be treated in a similar manner to the program itself: In cases where there is sufficient information to statically ascertain the potential for specializations, such checks could be be evaluated entirely at compile time. Where insufficient information exists to judge whether or not a given specialization is possible at run time, the extended PARTICLE system would have the option of incorporating the checks into the residual program and delay their evaluation until run time.[15] It should be noted that the use of the generalized specialization strategy offers a principled means of "turning computations into lookup tables" – computations applicable to a wide range and diverse set of classes of input can be mapped to an algorithm which simply classifies the input (via residual checks generated by the compiler) and dispatches to a set of specialized cases, each of which is associated with a far simpler (and frequently trivial) computation on a more restrictive and specialized program domain. Figures 5-18 and 5-19 illustrate several cases in which such a generalized specialization strategy can yield substantial performance advantages beyond those currently offered within PARTICLE.

5.6.3.2 Implementation Challenges

5.6.3.2.1 Introduction The primary challenges in extending PARTICLE in the manner described above lie in devising heuristics that allow the system to recognize those instances in which speculative specialization is likely prove most useful, and in the integration of those heuristics and associated speculative specialization mechanisms into the symbolic evaluation framework.

5.6.3.2.2 Devising Effective Heuristics for Performing Speculative Specialization The ability to delay decisions about the use of specializations until run time will prove beneficial only for cases in which the expected benefit of the specialization is sufficiently large to offset the run time overhead of checking the specialization conditions and the extra space cost of maintaining the specialized code. The expected benefit of a specialization is a function both of the performance benefit of that specialization when it is legal and of the frequency with which that specialization is actually applicable during an execution of the enclosing code. Heuristics aimed at addressing each of these components of the expected benefit calculation are discussed below.

There are a variety of techniques for improving the performance benefit of specialization effected by the system and for improving the ratio of that benefit to the performance cost of the run time checks. Specializations with high expected performance will tend to recognize and take advantage of potential for optimization at high (program and function) levels and opportunities for simultaneously performing several optimizations (e.g., specializing a sequence of statements with respect to some condition rather than individually specializing each statement with respect to that condition). By devising a framework in which identical predicates testing specialization conditions are successively merged and hoisted those to higher and higher levels of control flow, it may be possible for distributed low-level and individually unimportant speculative specializations to be combined into far more powerful and less expensive specializations. For example, several independent checks for the validity of a certain specialization condition can

[15]It is important to stress that within the scheme currently being discussed, while *checks* to see if a given optimization is possible may be delayed until run time, the fragments of code to be executed *if the specialization proves possible* will be specialized entirely at compile time: The strategy under consideration does not involve any potential for run time compilation.

```
{
        ...

        /*  p, q  pointers with unknown referents */

        *p = 0;
        *q = -1;

        for (i = 0; i < 1024; i++)
           {
             array[i] = *p; /*  system cannot rely on fact that *p has value
                                of 0 without knowing that p and q not aliased
                             */
           }
        ...
}
```

$\Rightarrow$

```
{
        ...
        /*  p, q  pointers with unknown referents */

        *p = 0;
        *q = -1;

        if (p == q)        /*  speculative specialization check inserted
                            by compiler */
                {
                for (i = 0; i < 1024; i++)
                   {
                   array[i] = -1;
                   }
                }
        else
                {
                for (i = 0; i < 1024; i++)
                   {
                   array[i] = 0;
                   }
                }
        ...
}
```

Figure 5-18: Within the code above, the system is unable to perform constant propagation upon the value of $*p$ due to the possibility that $*p$ is aliased with $*q$. Recourse to speculative specialization allows the system to create a run time conditional testing whether the two pointers actually are aliased.

```
int fact(int n)
{
        /*  n known to be in range 0-10 */

        if (n == 0)
                {
                return(1);
                }
        else
                {
                return(n * fact(n - 1));
                }

}
```

$\Longrightarrow$

```
int fact(int n)
{
        switch (n)
                {
                case 0:
                        return(1);
                case 1:
                        return(1);
                case 2:
                        return(2);
                case 3:
                        return(6);
                case 4:
                        return(24);
                case 5:
                        return(120);
                case 6:
                        return(720);
                case 7:
                        return(5040);
                case 8:
                        return(40320);
                case 9:
                        return(362880);
                case 10:
                        return(3628800);
                }
}
```

Figure 5-19: The ability to engage in speculative specialization offers the system a means of creating specialized code for each major class of expected input. In some cases, the computations

be combined into a single check, hoisted above an enclosing loop, and perhaps hoisted from the body of the called function to enclose many called functions in the body of the calling routine. Under certain conditions, it may even prove desirable to hoist specialization checks regarding input parameters to the program to the topmost level of the program in order to create speculative specialization contexts embracing the program as a whole.

The expected benefit of a given speculative specialization depends not only on the savings associated with the specialization itself, but also on the probability that the specialization performed will actually be used at run time. In certain cases, static information deduced during symbolic evaluation may permit the system to recognize that a potential speculative specialization or set of specializations are likely to be frequently exploited and are therefore likely to offer a high expected performance enhancement. For example, if it is known that a certain program quantity will take on one of two values at run time, specializations which individually exploit each these two cases can be safely and profitably performed by the system with the knowledge *every* time the original construct was executed, the time savings associated with at least one of these specializations will be realized. Figure 5-20 provided an illustration of a case where knowledge of the bounds on the possible values associated with a program quantity would permit the system to make use of speculative specializations with the confidence that the time saved by the specializations would more than outweigh the time overhead associated with the checks on specialization conditions.

Within the scope of the general guidelines given above there remains great latitude for devising practical heuristics to recognize and exploit opportunities for speculative specialization, and a great deal of work remains to be done in researching plausible means of approaching the problem.

5.6.3.2.3 The Integration of Speculative Specialization Mechanisms into PARTICLE

5.6.3.2.3.1 Introduction The implementation of speculative specialization can draw heavily on the capabilities of the existing PARTICLE system: The optimization of the speculative code can make direct use of the existing symbolic evaluation and specialization framework, and the heuristics needed to judge when speculative specializations will be advantageous have the opportunity to make use of a wide range of knowledge about program quantities and behavior. At the same time, support for speculative specialization requires considerable conceptual and mechanical extensions to the existing specialization framework. Some of the more important of these additions are discussed below.

5.6.3.2.3.2 Support for Conditional Contexts A fundamental prerequisite for the ability to take advantage of speculative specialization is the extension of the symbolic evaluator to allow it to make use of information concerning the truth or falsehood of a predicate while processing the body of the conditional associated with that predicate. (See section 5.5.3.) While such an capability can prove important as a general means of sharpening static knowledge in the absence of speculative specialization, it forms an absolutely essential prerequisite for the use of any mechanism permitting the symbolic evaluator to automatically create the speculatively specialized conditionals from knowledge of the speculative predicate. As discussed in section 5.5.3, such an extension requires the the addition of some non-trivial (but not overly

```
{
        ...

        a = sqrt(d);  /*  d known to be either 0, 3.1415927 or 100.0 */

        ...
}
```

$\Longrightarrow$

```
{
        ...

        if (d == 0.0)
                {
                a = 0.0;
                }
        else if (d == 3.1415927)
                {
                a = 1.7724539;
                }
        else
                {
                a = 10.0;
                }

        ...
}
```

Figure 5-20: Within the code fragment above, the system *knows* that one of the three specializations generated by the system will be used at run time for every execution of the residual code. As a result, the system knows that the "speculative" specialization process is guaranteed to yield performance benefits regardless of the actual patterns in which the generated specialization are used at run time.

complex) machinery to the existing PARTICLE framework – changes both to the mechanisms maintaining information about program Values and to the abstract domains used by the system. (In situations in which the system is considering the use of speculative specialization, it is particularly important that the abstract Value domains employed by the symbolic evaluation system are capable of expressing the truth or falsehood of the predicates to be created by the system and that the symbolic evaluator itself will have the ability to exploit this information within the conditional contexts.)

5.6.3.2.3.3 The Discovery of Opportunities for Specialization Within the existing PARTICLE system, opportunities for specialization are discovered in a purely operational manner: As code is executed, the system simply makes note of all cases in which the available data permits some reduction to be made within the existing program. It is not feasible to make use of such a methodology for discovering specializations when attempting to weigh the benefits of speculatively specializing some piece of code in a certain manner: There are far too many conditions to which code could be specialized to permit execution of the code with respect to each of them. Thus, while the actual *specialization* of a speculative piece of code can be performed by the existing symbolic evaluation and specialization machinery (augmented by the "conditional context" extensions to PARTICLE discussed above), the discovery of the *potential* for speculative specialization of a piece of code must rely on somewhat different mechanisms than are currently used to divine non-speculative specializations.

The potential exists for naively shifting *some* unknown compile time checks on program conditions to run time during the symbolic evaluation of that program: During symbolic evaluation, PARTICLE explicitly checks certain conditions that would permit specialization (e.g., the arguments to the "+" and "*" operations are checked for equality with zero whenever these operations are performed); compile time checks on such conditions could potentially be recorded, counted, combined, and turned into run time checks if judged sufficiently great in number or run time savings. On the other hand, other conditions permitting specialization (such as the knowledge that two pointers are not aliased) are *not* explicitly tested during symbolic evaluation; attempts to convert such implicit checks to explicit run time checks may prove much more problematic.

Section 5.6.3.2.2 sketched the basis for a uniform and conceptually simple manner of combining information about the potential for low-level specializations into predicates testing the opportunities for higher-level specializations. The method involved the successive combining and hoisting of predicates to allow the cost of testing the speculative predicate to be amortized over increasingly large areas of program code. Unfortunately, it is unclear how such a mechanism could be dovetailed with the symbolic evaluation framework: Hoisting predicates entails the creation of ever-larger conditional contexts requiring specialization – a change that yields increasingly large regions requiring specialization with respect to that conditional context. Moreover, the need to determine of the limits to which a predicate can be safely hoisted within program code may require the addition of considerable machinery to the current PARTICLE framework. At the least, the adoption of a hoisting-based methodology for enlarging the range of speculative specializations that can be performed by the system seems likely to require a substantial departure from the current symbolic evaluation framework. Substantial additional work is needed to delineate the means by which such repeated hoisting could take place and to flesh out its possible relation to symbolic evaluation.

While speculative specialization conceptually involves the shifting of checks for the legality of specializations from compile time to run time, the mechanisms used to implement this transfer of responsibility requires more complexity than might naively be suspected. In particular, to the degree to which checks for the legality of specializations are explicitly performed at compile time, they take place at the level of program *Values* (rather than explicitly manipulating *expressions*). In order to be incorporated as explicit predicates within residual code for the program, such checks must be rephrased as explicit expressions. Because all expressions within the "Simple C" processed by PARTICLE are free of side effects, it will be legal to create a residual predicate representing the speculative specialization conditional by simply representing each Value checked involved in that conditional in the conditional by the source code expression from which that Value was descended. (While such a practice has the potential for introducing many repeated subexpressions into the residual program, it is hoped that an aggressive post-processing compiler would eliminate the need to calculate such expressions more than a single time.) The time cost information computed during the execution of the predicate expression can provide a metric for judging the overhead associated with a speculative specialization conditional based upon that predicate. It is thus hoped that the addition of relatively minor machinery to PARTICLE will permit the reconstruction of residual predicate expressions from the underlying predicates carried out upon Values.

5.6.3.2.3.4 Conclusion The introduction of speculative specialization mechanisms into PARTICLE offers a means for greatly enlarging the scope and enhancing the power of PARTICLE's optimization strategy. The ability to exploit speculative specializations permits the system to aggressively optimize programs in a far greater set of circumstances than is currently possible and opens the possibility of performing very powerful specializations without the need for expensive and very high quality program analysis. The implementation of speculative specializations can make direct use of the specialization and analysis framework within currently offered by PARTICLE and of additions to that framework slated for future introduction into the system. At the same time, generalizing PARTICLE to make use of speculative specializations requires substantial additions to the current system at both the conceptual and implementation levels: The ability to make exploit speculative specializations requires the use of an alternative strategy for judging the benefits of specializations, the capacity for formulating, representing and exploiting knowledge that would appear beneficial for specialization, and the ability to effectively combine conditional tests within the specialization framework. Dealing effectively with each of these issues will require considerable design and implementation effort, and the distinctly different character of many of the analysis and optimization needs of the speculative execution framework calls into question the wisdom of attempting to fit such a framework into PARTICLE. It remains to be studied how neatly the speculative specialization mechanisms would dovetail with the current system. Because the possibility of performing speculative specializations offers such great potential for enhancing the strength of the specialization process and for providing a fundamental advance over existing optimization strategies, further research into the possibility of incorporating the capacity for speculative specialization into PARTICLE represents a high priority.

5.6.3.3 Run-Time Generation of Specializations

The discussion above has described a generalization the existing PARTICLE specialization methodology that permits the shifting from compile time to run time of predicates checking the validity of specializations created by the system. The ability to maintain the predicates within the residual code and to evaluate the associated conditions at run time greatly relaxes the constraints upon the specialization process and opens the possibility for exploiting powerful optimizations even in those cases where the available static information is insufficient to guarantee their validity or where program dynamics cannot guarantee their uniform applicability over all executions of the enclosing code. Unfortunately, speculative specialization is itself limited to allowing optimizations only within a certain class of situations: Within the methods discussed above, it is feasible to create specialized specializations only in contexts where *static data* regarding the program text indicates that such specializations are likely to offer high expected performance gains. A generalization of the speculative specialization methodology would permit this condition to be relaxed by allowing the shifting of the specialization process itself from compile time to run time. In particular, leaving open the option of resorting to run time specialization in instances where static knowledge was judged as insufficient to permit effective optimization of the code would permit PARTICLE to perform powerful specializations upon program code based on *dynamic* knowledge of the patterns of program data use. Consider, for example, the code fragment illustrated in figure 5-21. In this instance, static knowledge of program input is insufficient to permit compile time generation of specialized versions of the bessel function algorithm for each possible input. It would be straightforward for the system to create specialized clones of that algorithm for *any particular bessel function number* but without any prior knowledge about the distribution of user input to allow intelligent selection of the particular bessel function to specialize with respect to, it seems highly unlikely that a specialization of the general routine with respect to a particular bessel function number would offer much performance benefit within a typical execution of the program. At the same time, for a given execution of the program, only a *single* particular bessel function will be used – it is simply not possible to determine at compile time which function that will be. The ability to create specializations at run time would offer the system the option of generating residual code that at run time would construct a specialized version of the general bessel function routine for the particular bessel function desired during that run of the program. While the run time overhead associated with creating the specialized function may be substantial, the time saved by using the specialized function may more than compensate for this performance loss. More sophisticated examples of run time specialization might involve the adaptive creation of functions specialized to patterns of run time data (e.g., arguments to a function) that repeatedly recur during a computation. Although run time specialization represents a conceptually straightforward generalization of the current PARTICLE specialization strategy, it is a specialization technique that requires a wide range of mechanisms far beyond the scope of what is currently offered by PARTICLE. Although it is possible that such a strategy could be fruitfully combined with the existing specialization strategies (or some generalization thereof), the modifications needed to successfully carry out such a union lie far beyond the scope of this thesis.

5.6.3.3.1 Conclusions This section has sketched two means of generalizing the specialization strategies currently used within PARTICLE. Such generalizations offer the potential for

```
{
        ...
        /*  j unknown */
        j = ReadInt();

        for (x = 0; x < 10000; x += .01)
           {
            Plot(x, bessj(j, x));
           }
}
```

Figure 5-21: The code fragment above illustrates a case in which run time specialization can offer substantial performance advantages: At compile time, the symbolic evaluator lacks any knowledge about the range of values that can be adopted by *j* and is therefore incapable of specializing *bessj* to a specific *j*. Rather than attempting to create a specialized version of *bessj* at compile time, the system can instead delay the creation of such a function to run time. This specialized function could then be invoked within the inner loop instead of the original, unspecialized function.

dramatically expanding the effectiveness and scope of the optimizations performed by PARTICLE. Although PARTICLE currently only creates specializations where they are statically known to be legal, the generalizations discussed would permit the synthesis of specializations which could be used whenever run time conditions permit permit their legal application, and possibly allow for the run time creation of such specializations wherever the observed patterns of manipulated data are judged sufficiently skewed as to make such specialization beneficial. Such a generalized approach to program specialization allows powerful optimizations to be created even when static knowledge is too imprecise to guarantee their applicability, and offers a general-purpose means of realizing of the potential of program specialization to transform certain classes of computations into simple lookup tables. While an implementation of the generalized specialization strategy can take natural advantage of a range of PARTICLE's current capabilities, its realization will require the addition of a large amount of additional machinery to the existing framework, and some of the most important issues relating to the operation of a general system have yet to be worked out.

5.6.4 Conclusion

This section has sketched several means of enriching the set of optimizations performed by PARTICLE. Some of these extensions offered the potential for performing traditional compiler transformations within a framework offering access to much richer static information than is available in existing compiler systems. In contrast, section 5.6.3 presented a method for dramatically generalizing PARTICLE's specialization techniques to yield a system with capabilities even further removed from the sphere of traditional compiler optimization. For each optimization considered above, the expected performance benefit seems likely to more than repay the implementation cost of the extensions, but considerable additional study is needed to guarantee the desirability and feasibility of the proposed improvements

5.7 Improving the Quality of Specialization Decisions

5.7.1 Introduction

Previous sections have discussed extensions to PARTICLE that would help to improve the quality of static analysis performed by the system and broaden the set of optimizations that could be performed using the results of this analysis. The effect of each of these improvements is to enlarge the pool of specializations whose application will be considered by the system. This section turns to briefly consider the shortcomings of PARTICLE's specialization strategy and the potential for improving the quality of the choices made by PARTICLE as to which of this pool of possible specializations will *actually* be applied. (See section 4.4.)

5.7.2 Interprocedural Coupling between Function-Level Specialization Decisions

Section 4.4.3 discussed the failure of PARTICLE to take explicit account of the fact that specialization decisions for different functions are logical interconnected: PARTICLE specializes each function in isolation, and takes no explicit account of the fact that specializing a particular function in a certain manner is likely to have have implications for the costs and benefits associated with different manners of specializing other functions. PARTICLE currently makes no direct attempts to deal with such coupling between function specialization decisions, although the computation of the specialization decisions for each individual function are ordered in such a manner as to reduce the overall constraints on the specialization process. It seems probable that more sophisticated heuristics could be used to help make the specialization process more sensitive to the coupling between specialization decisions for different functions while maintaining computational tractability. On the other hand, it is not clear whether the benefits to be gained by recognizing the linked nature of patterns of specialization in different functions are sufficient to compensate the implementation effort needed to improve the performance of the current system. Research must be conducted in both of these areas.

5.7.3 The Use of Profiling Information in Specialization Decisions

PARTICLE currently makes use of simple static estimates of the frequency of execution of pieces of the user program. These estimates are used in attempts to judge the performance benefits associated with the use of particular specializations. Access to even coarse-grained profile information for "typical" inputs seems likely to offer the potential for substantially sharpening frequency estimates and thereby allowing the consistent application of more beneficial patterns of specialization.

5.7.4 Coupling Between Low-Level Specialization Decisions

As discussed in section 4.4, the specialization mechanisms within PARTICLE are ideally suited for operation in a system in which there is no coupling between the benefits of different specialization decisions. In particular, PARTICLE fails to take into explicit account the fact that the benefit associated with performing a certain optimization frequently depends logically on the presence or absence of other optimizations. Figure 5-23 illustrates two common patterns of specialization non-orthogonality. In the first example, the presence of one reductive optimization renders a range of other reductive optimizations unnecessary and worthless; ideally, a specialization system would recognize that in the presence of the higher level optimization the lower level optimizations offer no performance advantages and should not be treated as beneficial. The second example in figure 5-23 illustrates a similar phenomenon which can occur in conjunction with *non-reductive* optimizations. Non-reductive specializations can introduce code into the residual program that was not present in the original program source, and within the context of such transformations it is not only possible to have cases in which the *presence* of one optimization renders another specialization useless, but also instances where the *absence* of the non-reductive specialization can rule out the ability to perform many other optimizations. Section 4.4.4.1.2 presented some simple heuristics for associating more realistic cost information with optimizations in order to lessen the impact of failing to explicitly account for specialization coupling for reductive expression-level optimizations, but the problem can be considerably more serious in the context of non-reductive optimizations such as loop unrolling. For example, unrolling a program loop can allow powerful specializations to be carried out upon each copy of the loop body. Unfortunately, as illustrated in figure 5-23, the applicability and benefits of such subsidiary specializations are predicated entirely upon the existence of the specialization dictating the unrolling of the loop. The current PARTICLE framework has no means of representing or reasoning about this dependency; as a result, the subsidiary optimizations would be treated as orthogonal to the loop unrolling specialization, and the system would tremendously undervalue the benefit arising from the loop unrolling. The failure of the current system to take conscious account of the logical dependencies between specializations can potentially lead to significant decreases in the quality of the specialization decisions made by the system. While such failings are not fatal or truly grievous, the problems may be serious enough to warrant additional attention.

5.7.5 Conclusion

This section has discussed two areas in which PARTICLE's specialization strategy fails to recognize important couplings between specialization decisions. Further study is required to determine if the run time performance losses associated with PARTICLE's shortcomings are genuinely worth addressing, and additional work would be required to required to arrive at an alternative strategy that exploited additional information regarding the coupling between specialization decisions while maintaining acceptable limits upon the run time of the system.

```
{
        /*  b, c, d  fully known */
        a = (b * c) + (c * d); /* all subexpressions AND  entire expression
                                  capable of being collapsed */
}
```

Figure 5-22: All expressions within the code fragment can be completely evaluated at compile time. The specialization of higher-level expressions can make the reduction of lower-level subexpressions superfluous and without benefit, yet PARTICLE is incapable of explicitly reasoning about such links between the benefits of different specialization decisions.

```
{
        ...
        for (i = 0; i < 10; i++)
           {
           x[i] = exp(sqrt((double) i));
           }
        ...
}
```

$\Rightarrow$

```
{
        ...
        x[0] = 1.0;
        x[1] = 2.7182818;
        x[2] = 4.1132504;
        ...
}
```

Figure 5-23: The use of non-reductive specializations such as loop unrolling regularly results to situations in which the non-reductive specialization itself *permits* a wide range of other optimizations to take place. While the benefit associated with these additional optimizations is entirely predicated upon the presence of the non-reductive optimization, PARTICLE has no manner of representing or taking account of of such dependence when weighing possible patterns of specialization.

Chapter 6

Preliminary Performance Results

Previous chapters have discussed the motivations for PARTICLE and have sketched major aspects of the design of that system. Although the implementation of PARTICLE is an ongoing process, this chapter turns to briefly examine some preliminary measurements involving code processed by the system. Table 6.1 summarizes the performance improvement offered by specialized code generated by PARTICLE over the performance of desugared input code compiled by a highly aggressive optimizing compiler operating with all optimizations enabled. Before beginning a discussion of these results, it is important to mention some caveats relating to their interpretation.

6.1 Performance Measurements: Caveats

Although the performance results presented within this chapter have the potential for offering some insight into the benefits associated by symbolic evaluation, their interpretation demands a great deal of caution. In particular, the examples studied are uniformly of relatively small size and complexity; although two of the sample programs are popular benchmarks, the suite seems likely to offer resemblance to the large software systems most in need of *automatic* specialization. Moreover, many of the examples given were selected by virtue of the fact that they provided interesting opportunities for specialization. As a result, many of the examples are "embarrassingly specializable", and offer little resemblance to "common case code". As a result, it would be suspect to generalize in any direct fashion from the results given to the prospects offered by specialization for larger systems. Moreover, the measurements summarized in table 6.1 are drastically preliminary in that they have been taken without careful consideration of a number of important factors: The benchmarks have not been carefully normalized or studied for variations in the size of the abstract domains employed, the extent to which iteration should be allowed in analyzing program loops before loop abstraction is begun, the degree of unrolling allowed among loops, or the decision as to whether to enforce deterministic selection of Value identities when operating upon sets of preexisting Values. As a result, the results should be taken as highly qualitative in character, and as indicating only one possible speedup associated with specialization (rather than as a canonical expression of the benefits offered by such specialization).

Despite these caveats, the results of applying PARTICLE to the examples suggest a set of common patterns of benefits, and hint at the possibility for obtaining substantial performance benefits by exploiting specializations outside the reach of traditional compilers. Moreover, while larger software systems may not offer as much potential for speedup as the examples shown here, it is to be expected that certain *portions* of those larger systems *will* offer such opportunities.

Having briefly sketched some of the cautions to be exercised when considering the results shown below, the presentation now turns to consider the qualitative benefits offered by specialization for each of the example programs.

6.2 Specialization Experiments: Individual Summary of Results

Benchmark Name	Object of Specialization	Speedup Factor (vs Desugared)
Matrix Multiply (Sparse)	Program Text: Known Multiplicand Matrix	4.94
Matrix Multiply (Dense)	Program Text: Known Multiplicand Matrix	3.86
Dhrystone 1.1	Program Text	2.27
Poly Mandelbrot (Sparse)	External Input: Polynomial	1.84
Poly Mandelbrot (Dense)	External Input: Polynomial	1.25
Linear Interpolation	Program Text: Known Interpolation Function	1.77
Bessel Function Evaluation	Program Text: Fixed Bessel Fn #, domain restriction	1.52 (2.43)
TomCatV	Program Text	1.06

Table 6.1: This table summarizes the performance benefits obtained by applying PARTICLE to a small suite of programs. Column three of the table provides the ratio of the run time of the desugared input program to the specialized version of that input program. In the case of the Matrix Multiply and Polynomial Mandelbrot examples, the performance of a single program was sampled after being specialized with respect to input exhibiting different characteristics, and the table indicates the benefits obtained for each form of input data.

Table 6.1 summarizes the performance enhancements offered by PARTICLE for a series of sample programs. The input code was taken as input, desugared, and passed through PARTICLE to create a specialized residual program. Both the desugared and specialized programs were compiled using the Gnu C Compiler version 2.45 with all optimizations enabled (-O option set) and the run times associated with each of the programs were measured. The table shows the speedup offered by the residual, specialized program over the desugared version of that code.[1]

[1]As discussed in section 3.5.4 and shown in table 3.5.4, desugaring tends to affect the performance of the input code very little. As a result, all but the smallest percentage of the performance benefits indicated in table 6.1 arise directly from the specialization process rather than from the preliminary desugaring, and the performance benefits offered tend to vastly outweigh any minor loss in performance due to desugaring.

The speedups associated with specialization vary widely over the set of sample input programs. This section briefly surveys each of the benchmarks processed by PARTICLE and analyzes the benefits offered by specialization for each sample program; the next section turns to consider recurrent patterns of specialization seen within these benchmarks. For each sample program exhibiting substantial performance enhancement due to specialization, corresponding code fragments drawn from the original and residual programs will be shown to illustrate some of the effects of specialization on that program. For clarity, all dead and trivially useless code (assignments of a quantity to itself) have been removed from the specialized fragment within each such illustration.

6.2.1 Matrix Multiply

The matrix multiply benchmark is associated with the greatest performance boost offered by PARTICLE for any of the sample programs. The program consists of a use of a binary matrix multiply routine in a situation where the right hand multiplicand matrix is known at compile time. The benefits of specialization for the matrix multiple routine were judged for two different scenarios that differed in the composition of the known matrix. In particular, the matrix multiply routine was separately specialized with respect to one 3x3 dense right hand matrix in which all entries were non-zero, and with respect to a 3x3 sparse matrix in which 4 out of the 9 values were zero.

For the fixed matrices of both types, the optimizations performed by PARTICLE are centered around the creation of a specialized version of the matrix multiply routine which hard codes the composition of the right hand matrix. (See figure 6-1.) Within this specialized routine, the triply nested loop performing the multiply is unrolled, and constant propagation and subsequent constant folding are performed on the unrolled iterations of that loop. The constant propagation eliminates many data fetches from the residual program. For specialization with respect to a sparse matrix, significant amounts of constant folding can also be done. In particular, each element within the new array is formed by the dot product of a completely known vector with an unknown vector, and typically the system would be forced to accumulate the terms at run time. Howoever, in the case of the particular sparse example being considered, a large fraction of the intermediate multiply-and-add steps of the dot products are unneeded (because the multipliation yields a 0, and it is not necessary to the accumulate that 0 factor into the total) and can be omitted. (It is interesting to note that the opportunity for such an specialization would not be recognized by an offline specialization system, due to the inability of binding time analysis to discover specializations hinging upon knowledge of the particulars of the data being manipulated.) In the dense example, all known matrix entries are non-zero and thus offer less opportunity for constant folding and useless code elimination – a constraint that slows the dense matrix multiply down by a factor of 27

Figure 6-2 illustrates a recurrent characteristic of highly specialized residual code: the presence of large amounts of dead or useless code. Although aggressive constant propagation and folding and the collapsing of conditionals can render many assignments within the code unnecessary, PARTICLE makes no attempt to eliminate unneeded assignments from the code. As discussed in section 5.6.2.2, the delegation of this task to the optimizing compiler postpass is safe and typically leads to no performance loss. At times, however, the higher quality analysis performed by PARTICLE would allow the detection of code as dead which would not be recog-

```
 for (row = 0; row < 3; row++)
   {
     for (col = 0; col < 3; col++)
       {
         for (index = 0; index < 3; index++)
           {
             tmp[row][col] += (l[row][index] * r[index][col]);
           }
       }
   }
```

$\Rightarrow$

```
     tmp[0][0] = l[0][1] * -1;
     tmp[0][1] = l[0][0] * -1;
     tmp[0][1] = tmp[0][1] + l[0][1] * 4;
     tmp[0][1] = tmp[0][1] + l[0][2] * -1;
     tmp[0][2] = l[0][1] * -1;
     tmp[1][0] = l[1][1] * -1;
     tmp[1][1] = l[1][0] * -1;
     tmp[1][1] = tmp[1][1] + l[1][1] * 4;
     tmp[1][1] = tmp[1][1] + l[1][2] * -1;
     tmp[1][2] = l[1][1] * -1;
     tmp[2][0] = l[2][1] * -1;
     tmp[2][1] = l[2][0] * -1;
     tmp[2][1] = tmp[2][1] + l[2][1] * 4;
     tmp[2][1] = tmp[2][1] + l[2][2] * -1;
     tmp[2][2] = l[2][1] * -1;
```

Figure 6-1: The Specialization of the main loop of the Matrix Multiply sample program with respect to a right hand side multiplicand of known composition produces a straight-line execution sequence hard-coding knowledge about the elements of the right hand matrix. A sparse multiplicand yields substantial opportunities for constant folding and the elimination of useless assignments.

```
for (row = 0; row < 3; row++)
  {
    for (col = 0; col < 3; col++)
      {
        for (index = 0; index < 3; index++)
          {
            tmp[row][col] += (l[row][index] * r[index][col]);
          }
      }
  }
```

$$\Rightarrow$$

```
      row = 0;
      col = 0;
      index = 0;
      ( tmp[0][0] ) = 0;
      index = 1;
      ( tmp[0][0] ) = ( ( -1 ) * ( r[1][0] ) );
      index = 2;
      ( tmp[0][0] ) = ( tmp[0][0] );
      index = 3;
      col = 1;
      index = 0;
      ( tmp[0][1] ) = 0;
      index = 1;
      ( tmp[0][1] ) = ( ( -1 ) * ( r[1][1] ) );
      index = 2;
      ( tmp[0][1] ) = ( tmp[0][1] );
      index = 3;
      col = 2;
      index = 0;
      ( tmp[0][2] ) = 0;
      ...
```

Figure 6-2: The specialization of code fragments offering many opportunities for constant propagation or folding typically yields highly specialized computations embedded a matrix of dead or useless code. The removal of this dead code is an important job delegated by PARTICLE to the optimizing compiler post-pass.

nized as such by traditional compilation systems. As a result, while the large numbers of useless assignments present in highly specialized code are typically completely cleaned up during the compilation of the residual program, such assignments may occasionally not be recognized as useless and remain residual in the specialized program, yielding in some degree of performance overhead. (See section 5.6.2.2.)

6.2.2 Polynomial Mandelbrot

The polynomial Mandelbrot program was created to allow experimentation with fractal images generated by the calculation of conceptual variants of the Mandelbrot set. The area within the Mandelbrot set is computed by the iterative application of the mapping $x \leftarrow x^2 + c$ for points c on the complex plane. If the value of x diverges under repeated application of this mapping, the point c lies outside of the Mandelbrot set; otherwise, c is judged as belonging to that set. Computer calculations of approximations to the Mandelbrot set typically attempt to heuristically determine whether the repeated application of the above formula for given c would eventually lead to divergence by iterating the mapping a fixed number of times and seeing if the absolute value of x exceeds some threshold. The polynomial Mandelbrot program allows a generalization of the Mandelbrot calculation to permit the use of any iterative application of the form $x \leftarrow p(x) + c$, where $p(x)$ is some polynomial in x. The program accepts a number of parameters as input: The threshold above which a value of x will be considered as diverging, the number of iterations for which to iterate the mapping before comparing the absolute value of x to the threshold, the domain over which to compute the approximation to the Mandelbrot set and the step size between points at which to compute that approximation, and the polynomial $p(x)$ to be used in the iteration process.

PARTICLE was separately applied to the polynomial Mandelbrot program in order to specialize that program with respect to two polynomials exhibiting different densities. In particular, the program was specialized with respect to $p(x) = x^10 + c$ and $p(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ... + \frac{x^10}{10!}$. All other external input parameters were left unspecified. Table 6.1 shows the substantial performance enhancement associated with each of these specializations. Although there are small amounts of static data to exploit elsewhere within the program text, the specializations applied to produce the residual code in both cases center around the mechanism computating $p(x)$ for a given x (as illustrated in figure 6-3.). The code to evaluate a general polynomial steps through each term of the polynomial, computing the appropriate term in the series (x, x^2, x^3, etc.) and the contribution of that term to the total being accumulated. Specialization of the program with respect to an externally specified $p(x)$ creates a specialized version of this routine which unrolls the polynomial evaluation loop. The loop unrolling itself has a number of important benefits: it eliminates a large number of conditional checks, enlarges the size of basic blocks and lessens the performance penalties associated with branches, and eliminates the need to explicitly maintain loop induction variables. Most importantly, the loop unrolling allows the system to hard-code the identity of the polynomial to be evaluated into the polynomial evaluation routine by the consistent application of constant propagation and folding to the unrolled code. Such low-level optimizations eliminate many fetches and allow a shifting of a substantial amount of the run time computation to compile time.

This hard-coding permits the unrolling of the loop to calculate the polynomial and subsequent constant propagation and folding. As was the case for the specialization of the matrix

```
for (i = 0; i < MAX_COEFFICIENTS; i++)
  {
    /*  here, we are calculating and acumulating each term
        (i.e.,   1, x, x^2, x^3, x^4,...) */
    xNewReal += realFactor * PolyCoefficientArray[i];
    xNewImag += imagFactor * PolyCoefficientArray[i];
    {
      double dRealTmp;
      /*  calculate the next degree term (e.g., x^4 from x^3 * x) */
      /*   we have   (realFactor + i ImagFactor)(xReal + i xImag) */
      dRealTmp = realFactor * xReal - imagFactor * xImag;
      imagFactor = realFactor * xImag + imagFactor * xReal;
      realFactor = dRealTmp;
    }
  }
```

$\Rightarrow$

```
  xNewImag = 0.00000L;
  xNewReal = 0.00000L;

  dRealTmp = xReal;
  imagFactor = xImag;
  realFactor = dRealTmp;

  dRealTmp = realFactor * xReal - imagFactor * xImag;
  imagFactor = realFactor * xImag + imagFactor * xReal;
  realFactor = dRealTmp;

  dRealTmp = realFactor * xReal - imagFactor * xImag;
  imagFactor = realFactor * xImag + imagFactor * xReal;
  realFactor = dRealTmp;
  ...
  xNewReal = realFactor;
  xNewImag = imagFactor;
  dRealTmp = realFactor * xReal - imagFactor * xImag;
  imagFactor = realFactor * xImag + imagFactor * xReal;
  realFactor = dRealTmp;
  ...
```

Figure 6-3: Specialization unrolls the innermost loop of the polynomial mandelbrot program and permits limited constant propagation. Sparse polynomials can offer substantial opportunity for constant folding and dead code elimination. The example above shows the evaluation routine specialized to the polynomial $p(x) = x^10$. No accumulation of results is required until the final term of the evaluation, and many operations can consequently be eliminated from earlier iterations of the unrolled loop.

multiply routine, the constant propagation permits fetch elimination, while for sparse polynomials constant folding also allows the elimination of many useless multiply and accumulate expressions from the program; less constant folding is possible for the dense polynomial. Like the specialization of the matrix multiply example, the specialization of the mandelbrot program produced large amounts of dead and useless code that clutters the residual program but is easily removed during the compilation of the residual program. Table 6.1 shows the performance results of specialization for both sparse and dense polynomials and suggests that the character of the fixed data can have a first-order effect upon the size of the benefits offered by program specialization.

6.2.3 Generalized Bessel Function Evaluation

The previous example of program specialization involved the specialization of a program with a general polynomial evaluation routine to some specified input polynomial. The generalized bessel function example (drawn from a widely used library of numerical routines [43]) is similar in character but is associated with considerably more sophisticated parameterization. In particular, the routine *bessi* accepts as input two parameters n and x and returns the $I_n(x)$, where I_n is modified bessel function of index n.

In the modeling of many physical situations, the index of a bessel function being manipulated is dictated purely by the identity of the corresponding physical characteristic of interest within the model. As a result, particular uses of general bessel function routines are typically associated with *fixed* index n.[2] The specialization of *bessi* with respect to a known n therefore seems likely to represent a natural form of optimization. The index n parameterizes *bessi* in a somewhat involved manner, in accordance with a recursion relation specifying $I_{n+1}(x)$ in terms of a combination of $I_n(x)$ and $I_{n-1}(x)$; specializing the *bessi* routine with respect to a fixed n tends to offer highly substantial performance advantages by permitting the evaluation at compile time of considerable amounts of computation relating to this recursion relation.

The primary speedup given in table 6.1 represents the performance advantages offered by a specialization of the generalized bessel function routine to a particular n ($n = 2$) and a particular range of x ($x = 10.0 - 30.0$). This specialization yields a residual program in which a number of low-level optimizations have been performed. Although limited specialization is possible within the base-case *bessi0* routine, the most important optimizations take place upon the loop within *bessi* that computes $I_n(x)$ according to the recurrence relation among the I_k. (See figure 6-4.) The specialization of *bessi* permits this loop to be unrolled. The loop unrolling itself offers the standard advantages cited within previous examples, and the bounds imposed upon x allow the elimination a conditional from the loop[3]. In distinct contrast to earlier examples, while the unrolling permits moderate amounts of fetch elimination through constant propagation, there are few opportunities for constant folding aside from the ability to statically perform an implicit cast in each loop iteration.

[2]Note that the of a bessel function with a fixed index is a a recurrent pattern of use important enough to motivate [43] to offer not only the generalized bessel function routines within its library of mathematical functions, but also specialized versions of those routines to handle the common cases in which $n = 0$ or $n = 1$.

[3]Note that the conditional whose elimination is "allowed" by the unrolling (comparing j to n) would also be eliminated under *any* iteration-by-iteration analysis of loop behavior. Thus, although loop unrolling provides one means of recognizing that such a conditional is unneeded, it is by no means a precondition for making this discovery.

```
{
...
tox=2.0/fabs(x);
bip=ans=0.0;
bi=1.0;
for (j=2*(n+(int) sqrt(ACC*n));j>0;j--)
        {
        bim=bip+j*tox*bi;
        bip=bi;
        bi=bim;
        if (fabs(bi) > BIGNO)
                {
                ans *= BIGNI;
                bi *= BIGNI;
                bip *= BIGNI;
                }
        }
...
}
```

$\Rightarrow$

```
{
...
tempId4 = ( Specializedfabs0((double )x) );
tox = ( (float )( 2.00000L / tempId4 ) );
ans = 0.00000;

bi = bim;

bim = ( 1.00000 + ( ( 19.0000 * tox ) * bi ) );
bip = bi;
bi = bim;

bim = ( bip + ( ( 18.0000 * tox ) * bi ) );
bip = bi;
bi = bim;

bim = ( bip + ( ( 17.0000 * tox ) * bi ) );
bip = bi;
bi = bim;
...
}
```

Figure 6-4: The fragments above demonstrate the effect of specializing a routine to calculate the modified bessel function $I_n(x)$ to a known n and some possible range of x. The specialization permits an unrolling and subsequent specialization of the innermost loop, elimination of a call to the library function *sqrt* and a conditional.

The mix of low-level specializations performed within the loop of *bessi* is of a rather different composition than that seen within earlier examples, but accounts for the vast component of the speedup associated with the specialization process – an acceleration of the program by a factor of over 50%. The process of specialization creates a trivial specialized version of the *fabs* absolute value routine which simply returns its argument. PARTICLE currently has no capacity for inlining function calls (see section 5.6.2.3.), and the residual code contains many calls to the useless specialized *fabs* routine. Inlining is an optimization regularly performed by aggressive optimizing compilers, and the residual code was compiled with options instructing the compiler to inline the call to *fabs*. The performance of the resulting code with inlining enabled is giving in parentheses in table 6.1. As can be seen from the measurements, by taking advantage of preexisting specializations (in this case, the simplification of the *fabs* routine) inlining can ocassionally offer dramatic additional performance advantages to those provided by the current system.

6.2.4 Dhrystone 1.1

The Dhrystone program is a traditional and widely established benchmark for judging the performance of machine execution and the quality of compiler optimizations. Given the enormous popularity of Dhrystone as a measure of code performance and the particularly aggressive measures taken by some compilers to perform well on the benchmark[4], it seems that Dhrystone represents a natural candidate for examining the additional advantages offered by PARTICLE over traditional compiler analysis and optimization. The Dhrystone benchmark makes a single use of a C language memory allocation routine; the call is made during the initialization phase of the benchmark and is invoked only a single time during the execution of that program. Because PARTICLE does not currently provide support for the modeling of heap management routines, the program was very slightly modified to make use of a piece of memory allocated upon the stack rather than in the heap. Moreover, in accordance with the philosophy of allow PARTICLE access to as much of a system's code as possible (and in making all benchmarks as complete as open-ended as possible), code for the two library functions called by the body of the benchmark (*strcmp* and *strcpy*) was inserted into the file containing the benchmark code. All measurements were conducted upon this minor variant of the Dhrystone benchmark.

Dhyrstone performs no external input. As a symbolic evaluation system PARTICLE has the capacity for completely evaluating the entire program – because there is no dependence upon input, all values will be fully known, and the "symbolic" evaluation would in fact be a simple interpretation of the program. Such an analysis of the program would allow maximally precise recognition of opportunities for specialization and with generous allowance for unrolling could convert the entire program into a piece of straight-line code performing the output for the program. Unfortunately, the relatively long running time of Dhrystone prohibits the application of such a strategy: While it is possible in theory for PARTICLE to simulate all 50,000 iterations of the main program loop, the compile times would be extraordinarily long. PARTICLE is instead forced to perform a more rapid but less precise analysis of the program.

[4]The substantial inflation in reported Dhrystone 1.1 ratings due to Dhrystone-specific compiler "cheating" has been sufficient to call into question the benchmark's viability and continued use as a measure of program or machine performormance. In light of PARTICLE's dramatic acceleration of Dhystone, it should be stressed that PARTICLE contains absolutely no mechanisms designed with Dhrystone or any other benchmark in mind.

```
boolean Func2(StrParI1, StrParI2)
String30        StrParI1;
String30        StrParI2;
{
        REG OneToThirty         IntLoc;
        REG CapitalLetter       CharLoc;

        IntLoc = 1;
        while (IntLoc <= 1)
                if (Func1(StrParI1[IntLoc], StrParI2[IntLoc+1]) == Ident1)
                {
                        CharLoc = 'A';
                        ++IntLoc;
                }
        if (CharLoc >= 'W' && CharLoc <= 'Z')
                IntLoc = 7;
        if (CharLoc == 'X')
                return(TRUE);
        else
        {
                if (strcmp(StrParI1, StrParI2) > 0)
                {
                        IntLoc += 7;
                        return (TRUE);
                }
                else
                        return (FALSE);
        }
}
```

$\Longrightarrow$

```
int ( SpecializedFunc20 )(char  ( StrParI1[31] ), char  ( StrParI2[31] ))
  {
    auto int IntLoc;
    auto char CharLoc;
    auto int tempId21;
    auto unsigned int tempId22;
    auto int tempId23;
      {
        return(0);
      }
  }
```

Figure 6-5: The code fragments given above provide an example of the thorough specialization performed by PARTICLE on the Dhrystone benchmark. The first code fragment shows the original program code, which includes function calls and large numbers of conditionals. The second code fragment shows the same piece of code after processing by PARTICLE and subseqeuent dead code elimination. PARTICLE is capable of deducing the run time values associated with each program quantity being manipulated, and can completely performing the

As indicated by table 6.1, PARTICLE drastically improves the performance of the benchmark over the running time offered by the non-specialized code when compiled by a highly aggressive optimizing compiler. While the performance benefits gained within previous examples tended to arise from fairly well defined regions of specialization within the program, the sources for performance improvement within Dhrystone tend to be more finely distributed. Because the Dhrystone benchmark is of larger size than most other examples and the fact that the performance benefits of specialization within Dhrystone tend to arise from a wide set of different sources, the discussion of the advantages of specialization within Dhrystone will not attempt to characterize the individual loci of specialization within that program, but will typically seek instead to communicate the major features of the specializations performed therein.

6.2.4.1 Common Patterns of Specialization

The most important class of specialization applied by PARTICLE to Dhrystone lies in the high quality analysis and specialization of called functions based upon known features of the calling context. Even aggressive optimizing compilers typically offer no means of propagating rich information about values across procedure boundaries, are restricted to the application of very limited analysis methods and are incapable of specializing at the function-level, and the optimizations performed by PARTICLE within the Dhrystone benchmark lie considerably beyond the sphere of what traditional compilers are capable of exploiting. A concrete and particularly important example of the benefits accruing from function-level specialization can be seen in the specialization of Dhrystone's call to the *strcmp* library routine (a call in which the benchmark traditionally spends nearly one-third of its time). The *strcmp* function accepts two strings as arguments and compares those strings on a character-by-character basis. The routine then returns different values depending on whether the string associated with the first argument is lexicographically less than, equal to, or greater than that associated with the second argument. Within Dhrystone, the *strcmp* function is always invoked with two fixed and precisely known strings, and the call can be collapsed to a constant expression reporting the returned value. Unfortunately, recognizing the potential for such an optimization is difficult: The system must accurately model the content of strings, simulate the dynamic construction of the strings, propagate information about the strings as arguments to two called functions, and exactly simulate the iteration-by-iteration operation of the string comparison loop as it compares each pair of characters in the strings being analyzed. Each one of these tasks taken individually lies substantially beyond the sphere of the analysis offered by virtually any existing compiler, but all of these tasks are accomplished naturally and seamlessly within a symbolic evaluation framework. As a result, while PARTICLE can effectively eliminate one-third of the running time of Dhrystone as a straightforward consequence of its high quality analysis, traditional compiler analysis falls far short of being able to recognize the opportunities for optimization. Dhrystone offers a number of similar opportunities for completely evaluating function calls entirely at compile time. Because such evaluation has as a prerequisite the ability to accuratey simulate the run time flow of computation, such calls represent opportunities for specialization that are open for easy exploitation by a program specializer performing analysis based upon symbolic evaluation but which elude detection by the static analysis performed by even the most aggressive traditional compilers.

Dhrystone offers a large set of opportunities for performing fetch elimination by means of constant propagation; in some of these cases, the constant propagation depends upon high quality modeling of the contents of program arrays and pointers – modeling which is performed automatically within the PARTICLE framework, but which is frequently beyond the scope of compiler analysis.

Because of the high precision of the analysis carried out by PARTICLE's symbolic evaluation machinery, PARTICLE is capable of detecting a number of opportunities for eliminating branches within program code, simultaneously increasing the size of the basic blocks within the code being compiled, decreasing the overall size of the program by eliminating the code associated with branches known not to be reachable, and eliminating potentially disruptive branches from the path of execution.

Previous examples have stressed the performance benefits that can arise from the unrolling of program loops. Unrolling also plays an important role in the optimization of the Dhrystone benchmark, but offers only a modest performance boost (roughly 30%) on top of that gained from other forms of specialization. As in previous examples, loop unrolling proves beneficial both by virtue of eliminating branches and conditional checks within the residual program and by permitting the specialization of the unrolled loop iterations. Unrolling is a particularly natural optimization to perform upon the many loops in Dhrystone which can be statically shown to execute only one or two times.

6.2.4.2 Caveats

The section above has briefly characterized the primary sources of the performance enhancements which arise from the specialization of Dhrystone. Dhrystone is a fairly large benchmark and offers a variety of opportunities for high quality specialization at both high- and low- levels of the program. Moreover, the detection of many of the opportunities for specializations within Dhrystone requires extremely powerful analysis. As a result, Dhrystone is associated with a wealth of optimizations whose exploitation appears possible only within the context of a system such as PARTICLE. It seems natural to wonder to what degree the opportunities for specialization are representative of the potential for optimization within larger systems. Certain characteristics of the Dhrystone benchmark suggest that it is highly artificial and that the potential for specialization within Dhrystone is are unlikely to offer much resemblance to those seen in realistic contexts. In particular, certain aspects of Dhrystone seem to have been deliberately designed to test the "cleverness" of a compiler's analysis: The system offers an unrealistically large number of opportunities for constant propagation which are difficult to for traditional static analysis detect but whose exploitation is straightforward within PARTICLE. Even allowing for the use of specialization to reduce the constraints on the specialization of shared mechanisms, "real-life" software systems seem likely to offer much more modest amounts of static information for use in specialization. (For example, It seems likely that few (if any) realistic systems would offer the enormously lucrative potential for specialization seen within the the specialization of the *strcmp* routine described above.)

6.2.5 Linear Interpolation

The linear interpolation sample program makes use of a general interpolation routine that is parameterized by two arrays describing the points known to be on the function (and between which the function should be interpolated). The speedup measurement given in table 6.1 was based upon the specialization of the general interpolation routine with respect to a particular set of function data points. This specialization process yielded as output a new residual interpolation routine which "hard-codes" the identity of those data points and efficiently performs the interpolation process between those points. As reported in the table, the specialization process yielded an impressive speedup of 77% over the highly optiized version of the original code.

The specialization of the linear interpolation program leads to a proliferation of lower-level specializations in two major areas: In the routine initializg the arrays used to parameterize the general interpolation mechanism, and in the interpolation routine itself. Both of these routines are based upon a central loop whose unrolling forms the fundamental basis for all subsequent low-level specializations; as in previous examples, the unrolling of the loop also offers a number of direct benefits. While the specialization of both the initialization and interpolation loops benfit from loop unrolling, there are a number of differences between the patterns of subsidiary specializations performed in each situation. The subsequent specialization of each of these loops will be discussed in turn.

Within the initialization mechanism, the system is able to completely compute the contents of the program arrays at compile time; the unrolling of the initialization loop permits all table entries to be recorded directly within the residual code. As a result, specialization converts an abstract loop involving calls to external functions in the C language math libraries into a simple sequences of assignments inserting the appropriate constant values into their respective array elements.

The central interpolation routine operates by stepping through each pair of function points and comparing the x coordinates of those points to the x coordinate of the point whose value is to be interpolated. The specialization of this routine benefits enormously from the complete unrolling of the loop; the loop unrolling brings all of the benefits normally attendant upon such a transformation, but also allows a large number of lower-level specializations. In particular, the unrolling of the central interpolation loop allows the residual program to hard-code the (x, y) location of each of those points within the specialized interpolation code, eliminates all fetches from the parameterized arrays, and voids the need to perform any parameterization whatsoever. In contrast to previous examples, despite the fact that PARTICLE has substantial knowledge of the static data being manipulated, there PARTICLE takes advantage of few opportunties to perform much constant *folding*. However, PARTICLE's inability to perform constant folding in this case is due less to an inherent difference in the code between the current sample program and previous examples than it is to a combination of a shortcoming in PARTICLE's repertoire of specializations coupled with an accident in program phrasing: The *return* expression offers an opportunity to perform constant folding by combining two statically factors within a multiplicative term in which the third factor is not fully known. Unfortunately, PARTICLE current has no capacity for performing constant folding within such a context, and fails to recognize this opportunity for eliminating a run time computation. The postpass optimizing compiler used in compiling the original and residual C code is also unable to recognize the opportunity for constant folding within this situation; were either PARTICLE or the optimizing compiler capable of discovering the potential for constant folding in situations such as this one, the performance of the specialized code would be correspondingly greater.

In summary, the specializations performed within the linear interpolation sample program are rather simple in character: The vast proportion of the speedup due to specialization stem

```
void SetUpTable(double xx[], double yy[], int n)
{
  int i;
  xx[0] = NEGATIVE_INFINITY;
  for (i = 0; i < n - 1; i++)
    {
      xx[i] = log(i + 1);
      yy[i] = exp(sqrt(i));
    }
  xx[n-1] = INFINITY;
  yy[n-1] = yy[n-2];
}
```

$\Rightarrow$

```
void ( SpecializedSetUpTable0 )(double  ( xx[] ), double  ( yy[] ), int n)
{
  auto int i;
  auto double tempId0;
  {
    ( xx[0] ) = -1.00000e+06L;
    ( xx[0] ) = 0.00000L;
    ( yy[0] ) = 1.00000L;
    ( xx[1] ) = 0.693147L;
    ( yy[1] ) = 2.71828L;
    ( xx[2] ) = 1.09861L;
    ( yy[2] ) = 4.11325L;
        ...
    ( yy[6] ) = 11.5824L;
    ( xx[7] ) = 2.07944L;
    ( yy[7] ) = 14.0940L;
    ( xx[8] ) = 2.19722L;
    ( yy[8] ) = 16.9188L;
    ( xx[9] ) = 1.00000e+06L;
    ( yy[9] ) = 16.9188L;
  }
}
```

Figure 6-6: The linear interpolation mechanism is associated with two arrays initialized upon program startup. The initialization process can be simulated exactly during program analysis, permitting the array contents to be calculated entirely at compile time. Specialization permits the results of such calculations to replace the original initialization mechanisms.

```
for (i = 0; i < n - 1; i++)
  {
    if (x <= xx[i + 1])
      {
        return((yy[i + 1] - yy[i]) * ((x - xx[i]) / (xx[i + 1] - xx[i])));
      }
  }
```

$\Rightarrow$

```
  ...
  if (x <= 0.693147L)
     return(1.71828L * ( x / 0.693147L ));
  if (x <= 1.09861L)
      return(1.39497L * ( ( x - 0.693147L ) / 0.405465L ));
  if (x <= 1.38629L)
      return(1.53898L * ( ( x - 1.09861L ) / 0.287682L ));
  if (x <= 1.60944L)
      return(1.73682L * ( ( x - 1.38629L ) / 0.223144L ));
  if (x <= 1.79176L)
      return(1.96741L * ( ( x - 1.60944L ) / 0.182322L ));
  ...
```

Figure 6-7: The code fragment above indicates the results of specializing the interpolation routine with respect to a set of known data points. The specialization process unrolls the loop stepping through the data points and hard-codes the knowledge of the data points in the specialization of each unrolled iteration. Although PARTICLE eliminates many data fetches by means of constant propagation, the system fails to discover an opportunity for performing limited constant folding within the unrolled loop.

merely from the elimination of fetches and branches, checks of iteration conditions, and maintainance of loop induction variables by loop unrolling. Nonetheless, the consistent application of these specializations yields an impressive 78% speedup of the entire program. Although traditional compilers commonly include such specializations within their optimization repertoire, they lack the capacity for high quality analysis needed to discover the opportunities for performing such specializations within the sample program being discussed. In particular, even the most aggressive optimizing compilers generally lack the ability to evaluate external routines on statically known data (as is done for the mathematical functions within the initialization loop), to model the values associated with the individual elements of arrays, to propagate rich information about program data across procedure boundaries, and to relax specialization constraints by creating distinct specialized versions of a general-purpose routine. Such capabilities are provided transparently within PARTICLE's symbolic evaluation framework and allow PARTICLE to perform considerably more aggressive optimization than is possible using such traditional compilation systems.

6.2.6 TomCatV

The *TomCatV* benchmark is a C version of code drawn from the SpecMark 1.2 benchmark suite, and generates a mesh using Thomson's solver. The program is a stand-alone scientific application that accepts no external input and makes no non-trivial calls to external routines except to output data on program behavior. The main loop of the program is a relaxation routine which is iterated until convergence is reached or until a certain maximal number of loop iterations have elapsed. For the specialization experiment whose result is reported in table 6.1, a slight modification of the original program was used: The original system manipulated a number of double-precision arrays of size 257 by 257. The simulation of that system required extravagant amounts of compile time memory due to the extremely low-level models of program state manipulated by PARTICLE, and the computationally intensive loops cycling through the elements of the arrays were associated with huge running times. In order to obtain speedup measurements within some reasonable amount of time and space usage, the modified version of *TomCatV* processed by PARTICLE is associated instead with arrays (and associated loops) of far more modest size (17 by 17 rather than 257 by 257). In addition, the a number of *goto* statements expressable as more structured constructs were replaced by those constructs.

Like *Dhrystone*, *TomCatV* accepts no external input and is guaranteed to terminate, and PARTICLE is thus in principle capable of exhaustively simulating the entire run time execution of the program and gathering complete and maximally precise information on the set of legal optimizations that can be applied to the code. Unfortunately, the interpretive overhead associated with symbolic evaluation is far too great to make such a method practical. PARTICLE is instead forced to approximate the behavior of the program in a much more coarse but rapid manner.

PARTICLE performs very limited specialization within *TomCatV*: The system propagates constants to the loop termination conditions and performs many type casts at compile time. The system was prohibited from performing loop unrolling because of fears of a space explosion within nested loops and the perception that such unrolling would offer few performance advantages or would lead to excessively long compile times. The main body of *TomCatV* performs a relaxation algorithm involving several arrays. In previous examples the most effective specializations were been carried out on code with respect to some data fixed during the evaluation of that code (e.g., a polynomial evaluation routine with respect to an array containing the coefficients for a polynomial, or a string comparison routine with respect to some fixed strings being scanned). Within *TomCatV*, the central data arrays are themselves changing with each iteration of the main loop, and there are few opportunities for specialization of the central loop with respect to any data fixed between iterations of that loop. The conditions for the specialization of that loop could be relaxed by unrolling the loop and specializing each iteration separately, but such an approach effectively requires an expensive simulation of many different iterations of the central loop of the program (and is therefore tantamount to an interpretation of much of the program itself). Thus, while *TomCatV* in principle offers many opportunities for specialization arising from the fact that all data originates within the program itself (in a sense, because "all external input is known"), compile time constraints coupled with a lack of *fixed* static data prevent the effective exploitation of virtually all of this potential knowledge about the program data.

In a dramatic contrast to other examples, the *TomCatV* benchmark exhibit very little benefit from specialization with PARTICLE – processing by the system accelerates the benchmark by only a marginal factor of 6% over the original program. *TomCatV* thus serves as an indication that while PARTICLE's high quality analysis and specialization are capable of performing high quality specialization in a wide variety of contexts, they are by no means guaranteed to offer performance enhancement above that allowed by highly optimizing compilers. Moreover, the *TomCatV* example stresses the current importance of user judgment in setting limits on loop unrolling and the number of iterations of a loop to simulate prior to the abstraction of that loop. Ideally, the space throttling possible within the specialization phase of PARTICLE would preclude the need for the user to consider the space consequences associated with unrolling. Unfortunately, the current design of the system offers no elegant means for reasoning about specialization decisions in the presence of non-reductive optimizations such as loop unrolling. As a result, even if the currently designed specialization mechanisms for tightly throttling cost/benefit decisions were fully in place, until a more satisfactory specialization strategy is adopted the loop unrolling decisions would still lie in the hands of the user. (See section 5.7.4.)

6.3 Specialization Experiments: Initial Interpretations

This chapter has presented preliminary timing measurements from a number of sample programs processed by PARTICLE. As stressed above, these performance measurements are extremely preliminary and purely qualitative in character; moreover, both the speedups and their underlying causes vary widely between sample programs. Nonetheless, there are some recurring patterns within the results that merit discussion.

6.3.1 Recurring Patterns of Specialization

This section discusses the specializations offering the greatest performance advantage within the examples above, in the hopes of distilling some sense of what aspects of the specialization process are most important; a subsequent section turns to consider what components of PARTICLE's *analysis* strategy offer the greatest benefits for the system. As might be suspected, higher-level specializations tend to yield the greatest performance enhancement in compiled code: in general, the higher the level of the specialization, the larger number of subsidiary specializations it can engender by virtue of reducing the number of contexts joined at the meet point for the original mechanism. The three most crucial forms of specializations seen within the examples above are specialization at the program level (partial evaluation of a program with respect to some external input to that program), and at the function-level, and at the statement (or loop) level.

As discussed in section 3.1.3.5, the specification of a small bit of information concerning program values is far more likely to be valuable at higher levels of specialization, for the ramifications are likely to be much greater. Information concerning external program input tends to have global ramifications for program behavior and for the data manipulated within the program. As a result, specialization at the level of the entire program tends to allow a proliferation of low-level specializations and consistently offers substantial performance advantages.

Specialization at the function-level seems essential for effective program specialization: Aggressive specialization relies strongly on the ability to propagate information about program data across function boundaries: Without recourse to such an ability, a program specialization system would be unable to detect the potential for the *vast* majority of the optimizations exploited within the examples above. Unfortunately, without function-level specialization, such interprocedural propagation of information is likely to prove fruitful due to the implicit data flow meet point present at the entry to every function. Moreover, because most abstract mechanisms within program are realized as functions (or collections of functions) specialization at the function-level is a prerequisite for reducing the performance cost associated with the use of abstraction and generality in program design. As a concrete illustration of this need, consider the examples above: The sample programs above repeatedly made use of general mechanisms encoded as functions (e.g., the matrix multiply routine). Without the opportunity to specialize such mechanisms to particular uses of those mechanisms, virtually none of the specializations performed above would be legal (except for very small programs in which only a single use of the general mechanisms was present, or larger programs exploiting only a small sliver of the functionality of a general mechanism). The sample programs thus strongly attest to the value of function-level specialization.

Specialization of program loops by unrolling consistently played an important role within the specialization of the example programs. As briefly mentioned above, loop unrolling is associated with substantial direct and indirect benefits: The direct benefits of loop unrolling stem from the elimination of branches from the program code (yielding both larger basic blocks for exploitation by the postpass compiler as well as fewer opportunities to disturb processor pipelines), the ability to elide checks on loop iteration (or termination) conditions, and the frequent opportunity to entirely avoid the maintainance of loop induction variables. In general, however, the greatest benefits offered by loop specialization are indirect ones: Loop unrolling often opens the opportunity for performing a wide range of lower-level optimizations within each specialized iteration of the loop. This benefit was seen repeatedly and dramatically within the examples, as loop unrolling permitted enormous amounts of "hard coding" to take place within each specialized iteration, almost always allowing for constant propagation and folding and sometimes permitting the collapsing of conditionals. This "hard coding" of each iteration then frequently responsible for a dramatic decrease in the run time of the residual program. Loop unrolling thus frequently represents a crucially important form of specialization for increasing program performance.

It should be stressed that despite the many advantages it offers, the unrolling of a loop is *not* always beneficial for a program: For example, a study of the benefits offered by specialization to the Dhrystone benchmark revealed that while the specialization of the string copy routine *strcpy* to two particular, fully known strings allowed the elimination of half of the data memory references performed within that code (effectively converting a string *copy* into a series of individual insertions of known bytes into the target string), the unrolling actually *decreased* program performance, almost certainly due to a worse instruction cache hit rate.

6.3.2 Analysis Requirements of High-Quality Specialization

The previous section attempted to characterize those classes of specializations that were responsible for the greatest performance advantages within PARTICLE. This section turns to consider the aspects of program *analysis* which proved the most important within the specialization of the examples above.

The discussion above cited the crucial role played by function-level specialization in sample programs; the ability to perform such specialization requires the use of high quality interprocedural data flow analysis, and it is frequently desireable to perform high-fidelity simulation of control flow within the function being specialized. Each of these requirements is briefly discussed below.

In order to permit effective function-level specialization, the system must offer some means of collecting and maintaining the information with respect to which a function is to be specialized. The specialization of a function with respect to a particular call site to that function requires the ability to propagate knowledge of the state obtains at that calling site to the program to be specialized. The examples presented within this chapter underscore the importance of collecting *rich* contextual information for use during interprocedural analysis: Many function-level specializations were conducted relative to particular array contents – information that is rarely (if ever) maintained even *locally* within program analysis. The collection of high quality interprocedural data flow information allows aggressive and highly beneficial function-level specialization.

In addition to indicating the crucial place of interprocedural analysis, the patterns of specialization seen within the examples studied above repeatedly bear witness to the importance of PARTICLE's capacity to perform high-fidelity simulation of the run time control flow (particularly within program loops and – to a lesser degree – function calls. In particular, the high quality iteration-by-iteration (or recursion-by-recursion) simulation of program loops can allow the extraction of *far* more precise information about the effects and operation of such loops than is possible within a minimal fixed point (MFP) analysis framework, and in general can permit the true shifting of computations involving general control flow from run time to compile time by allowing the analysis machinery a means of closely emulating the patterns of run time control made during a computation. For example, even in a system which performed first-class propagation of interprocedural dataflow information and modeled the contents of program arrays (and thus strings in C) but lacked a means of performing iteration-by-iteration loop analysis, a *strcmp* routine comparing two known strings could never be recognized as returning the correct return value or as reducible to a constant – as discussed in section 4.3.3.2.2.3, the fixed point iteration process would "mix together" knowledge of the program state at many different points during the execution of the loop, and would prevent the system from recognizing that the pointers are advancing in lockstep across their respective strings (and therefore that one pointer always points to the same character as the other pointer). Similarly, a system offering only MFP analysis could never shift from run time to compile time the interpretive overhead associated with the application of a print formatting routine to a known format string, and would never be capable of evaluating a substantial computation at compile time (such as an FFT as applied to a nown piece of data) at compile time.[5] In the absence of high quality loop simulation, a program specialization system will frequently lose large amounts of static information about program quantities when encountering even short loops, and will be generally be unable to efficiently make use of static information discovered during program analysis.

The examples above consistently suggest the importance of one additional aspect of high quality analysis: The modeling of program arrays. The substantial majority of the specializations above were carried out relative to an array specifying the data to which to specialize the general mechanism, and it is common for general mechanisms to be parameterized with respect to a vector of information rather than a simple series of discrete pieces of data. Forfeiting the ability to maintain information about the contents of program arrays during analysis would greatly restrict the domain of possible candidates for specialization. Moreover, because program loops frequently operate on arrays, loop unrolling and the capacity for maintaining information on the contents of program arrays can prove especially complementary, with each optimization reinforcing the benefits associated with the other. The sample programs considered above offer many examples of this phenomenon.

The examples above illustrate the benefits of a final capacity that can prove particularly useful during program analysis. Like the ability to perform high quality simulation of program loops, the capacity for performing calls to library routines at compile time allows the program to take an important step towards a general capability for shifting general computation from run time to compile time. While the ability to elide calls to library routines did not appreciably accelerate the code within the examples above, the unimportance of this optimization for the given sample programs stems more from a general lack of use of library routines within those programs rather than from any shortcoming in the benefits offered by this ability. The examples above (particularly *Linear) Interpolation* provide enough demonstration of the utility and power of this capacity to suggest that it would prove highly beneficial within programs that do rely more heavily upon library functions.

This section has attempted to characterize those aspects of program analysis most important for allowing the discovery and application of high quality specializations. Although the evidence is grossly incomplete, it would appear that high quality analysis of program control flow and arrays, and the use of full-bodied interprocedural data flow analysis can contribute greatly to the quality of specializations performed within the residual program.

[5]Although it is frequently possible to accomplish iteration-by-iteration analysis of a loop within an minimal fixed point framework by unrolling a loop prior to its analysis and then analyzing the unrolled version of that loop, such a technique tightly couples the quality of program analysis to the *optimizations* performed on the program being analyzed: The system fails to recognize that there may be situations in which it is not desirable to unroll a loop (e.g., due to space considerations), but where it is desired that the loop be finely simulated.

6.4 Conclusion

This chapter has examined the advantages offered by the application of PARTICLE to a small set of programs. The sample programs are uniformly rather small, and the performance measurements taken upon those programs are only suitable for a purely qualitative examination of the advantages offered by specialization. Nonetheless, the sample programs offer some preliminary hints as to what aspects of program analysis and optimization prove most valuable when specializing a function, and lend a rough feel for the advantages offered by the specialization process. In particular, the examples suggest that multi-level specialization can offer the potential for *dramatic* speedups in certain ranges of situations, and that specialization at the program, function and loop levels can prove particularly important in fostering good program performance. Moreover, the character of the specializations discovered within the sample programs suggest that the use of high quality program analysis forms a necessary precondition for the exploitation of many (and perhaps most) specializations. In light of these highly preliminary results, it appears that PARTICLE's use of high quality program specialization based upon symbolic evaluation places the system in a strong position to valuably boost the performance of a wide variety of software systems. Much work is needed to investigate the extent and character of the benefits offered by program specialization, but the preliminary results tentatively suggest that the approach is likely to be quite beneficial, and may offer a valuable and profitable extension of program optimization techniques to regions of program behavior to which their application has not previously been possible.

Chapter 7

Conclusions

This thesis has discussed the design of PARTICLE, a program specializer aimed at boosting the performance of programs written in low-level and imperative languages. Preliminary results tentatively suggest that PARTICLE's overall strategy of performing multi-level specialization based on high quality program analysis offers the potential for exploiting a class of important optimizations whose discovery lies well beyond the range of traditional program analysis strategies. At the same time, implementation of PARTICLE has exposed serious shortcomings in the system design. Previous chapters have discussed some of the strong and weak points of PARTICLE, but frequently such assessments have been scattered throughout individual chapters or only mentioned in passing. In order to impart a sense of the general benefits and shortcomings of PARTICLE's design, this section will review some of the major positive and negative aspects of the system. For the sake of brevity, this list is confined to discussing *design decisions* within PARTICLE that are classified as successes or failures based upon the current perception as to how well they advance the general goals of the system.

- **Successes**
  - **Higher-Level Specialization** Specializations at higher levels of a program can greatly reduce the constraints upon low-level optimizations and can thereby permit the creation of specialized mechanisms that are highly tuned for use in particular ranges of contexts, and PARTICLE's capacity for performing specializations at the program and function-levels allows for dramatic program performance boosts within the residual program. Program-level specialization with respect to partially specified external input frequently leads to substantial (and sometimes drastic) performance enhancement throughout the body of the program being specialized. Function-level specialization is absolutely essential to program specialization: Most abstract mechanisms within languages are at least partly encapsulated within functions or groups of functions, and the ability to effectively specialize a program's mechanisms to particular patterns of use thus generally relies directly upon a capacity for performing function-level specialization. Specialization of functions can substantially reduce the costs associated with good software engineering and in general provides a powerful means of exploiting the substantial amounts of static information frequently found at call sites [19].

- **New Specialization Methodology** The two-phase character of PARTICLE's kernel allows the system to make far more informed choices between possible patterns of high-level specialization than were possible in previous specialization systems. The decisions to delay all specialization decisions until data concerning program behavior is complete and to perform explicit weighings of the costs and benefits associated with different high-level specialization options provide PARTICLE with a principled means of gaining substantial performance boosts from program specialization while maintaining tight control over the space expansion frequently associated with specialization.
- **Arbitrarily Precise Control Flow Analysis** The symbolic evaluation machinery associated with PARTICLE is capable of making a wide variety of tradeoffs between extremely fine-grained simulation of program control flow (in which loops are simulated on an iteration-by-iteration basis) and faster but coarser forms of control flow analysis (where loops are typically analyzed by means of immediate resort to fixed-point iteration). The capacity for simulating program control flow in manner mirroring its run time execution permits the system to collect far more precise static knowledge than is possible within traditional analysis frameworks, and allows the system to shift vastly more computation from run time to compile time than would be possible within such systems.
- **Interprocedural Analysis** As noted above, function-level specialization is plays a central role in effective program specialization, but the ability of a system to carry out specialization at the function-level itself depends crucially on the quality of the interprocedural analysis performed within the system. Interprocedural analysis makes available information about the calling contexts of a function that permit effective specialization of the called function to particular use patterns of that function. PARTICLE's capacity for performing high quality interprocedural analysis proved highly valuable during the processing of the sample programs and appears to represent an important asset of the current system.
- **Modeling Elements of Arrays** The examples presented within chapter 6 consistently suggested the importance of high quality modeling of array elements: The specialization of routines with respect to the contents of program arrays can be extremely valuable, and will likely be a fairly common need within the domain of program specialization.
- **Analysis Extensibility** One of the central component of PARTICLE's philosophy is the desire to provide the user with the option of exercising maximal control over the character of the specialization process – particularly with respect to the tradeoff between specialization quality and analysis time. This tradeoff itself involves a number of parameters which can be set independently by the user to focus certain aspects of static analysis on those portions of a program or those programs where they is most needed. Rather than performing a single, monolithic form of analysis, the system is capable of applying any of a wide set of different analysis techniques, in accoradance with the user's judgement as to what form of analysis is most appropriate.
- **A Research Framework** PARTICLE provides a general and arbitrarily precise *framework* for studying characteristics of static information availability, program

analysis and specialization. In particular, the system permits investigations into the performance benefits associated with different specialization and analysis strategies, and also provides a natural framework in which to undertake a more basic and open-ended study of the general availability, origin, variability, and character of static information within programs. It is hoped that the ability to survey the advantages and shortcomings associated with different specialization and analysis techniques will provide feedback as to the strengths and weaknesses of the current PARTICLE system and will eventually allow a reassessment of and ultimately changes to the design of that system to permit much more economical forms of program analysis without substantially sacrificing the quality of the specializations supported.

- **Weaknesses**
  - **Lack of Constraints on Function Calls** In general, the analysis performed by PARTICLE transparently crosses a procedure boundary whenever the symbolic evaluator encounters a function call within the program being analyzed. While this practice results in an extremely full-bodied form of interprocedural flow analysis, it leads to huge analysis times and renders hopeless the symbolic evaluation of all but relatively small programs. PARTICLE currently provides no means of throttling the precision with which function calls are simulated; even if the user wishes to make use of the most rapid analysis possible, the system will nonetheless adhere to its practice of simulating the behavior of function calls to the greatest precision possible. As a result, program analysis within the current version of PARTICLE is generally associated with an unaccpetably high mandatory base-line cost.
  - **Low-Level Simulation of Memory Model Contents** PARTICLE currently maintains explicit approximations to program state that are extraordinarily fine-grained and model the store on a byte-by-byte basis. While such precise approximations to the store allow enormous fidelity and precision in simulating low-level pointer operations, they are associated with a substantial space and time overhead. Moreover, while the system gives the user precise control over the richness of the value approximations maintained within the system, no such flexibility is offered over the precision of the state approximation. Like the practice of offering nearly-unrestricted simulations of function calls, the rigid and mandatory maintainance of a high quality state representation currently exacts a large and inflexible performance cost upon system operation and demands a redesign.
  - **Emphasis on High-Quality Abstract Domains** The design of PARTICLE emphasized the use of high quality abstract domains as a means of gathering extremely precise information concerning program behavior. While rich abstract domains can play an important role within certain ranges of circumstances (e.g., to bound knowledge concerning important quantities such as loop indices or pointer referents), it remains to be seen whether they offer as much potential for improving the quality of program analysis as was originally anticipated. Given the repliminary results of applying PARTICLE to a small suite of sample programs, it appears that the emphasis upon rich abstract domains as a means of achieving more precise analysis may have been somewhat misplaced: Other parameterized aspects of the analysis seem considerably likely to be more important for improving the quality of the specialization

process than are adjustments to the richness of the value domains. It is anticipated that studies of the benefits offered by analysis on realistic programs may eventually lead the system to sacrifice the capacity for performing high quality value analysis in order to greatly accelerate the speed of that analysis.

7.1 Immediate Directions

Chapter 5 provided intensive discussion of a number of ways in which the current system could be fruitfully modified or extended, either to save compile time resources or to extend system functionality. While it is likely that a number of the suggestions from within the chapter will eventually be incorporated into PARTICLE, implementation will first concentrate on stabilizing the current system and addressing some of the more glaring lacunae discussed in sections 1.2 and 5.2. It is anticipated that attention will then turn to the task of drastically reducing the time and space overhead associated with the current system by throttling the procedure call mechanism (through the use of a strategy similar to that detailed within section 5.3.3.1.) and by substantially reducing the mandatory precision but increase the flexibility and expected performance of the state approximation abstraction. Throughout this process, the PARTICLE will be used to study the character and availability of static data within sample programs, and to examine the specialization benefits offered by different classes of specialization and program analysis. It is hoped that as PARTICLE becomes increasingly capable of processing large program files, the system will acquire the capacity to study the character of static information and program specialization within more realistic programs and that such a study will help to serve as a close guide in the further development of PARTICLE.

Bibliography

[1] S. Abramsky and C. Hankin. An introduction to abstract interpretation. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 9–31. Ellis Horwood Limited, Chichester, West Sussex, England, 1987.

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

[3] A.W. Appel. A efficient program for many-body simulation (or,cray performance from a vax). Technical Report CMU-CS-83-118, Carnegie Mellon University, Pittsburgh, PA, 1983.

[4] A.E. Ayers. *Abstract Analysis and Optimization of Scheme.* PhD thesis, Massachusetts Institute of Technology, September 1993.

[5] A. Berlin. A compilation strategy for numerical programs based on partial evaluation. Technical Report 1144, MIT AI Laboratory, 1989.

[6] A. Berlin and D. Weise. Compiling scientific code using partial evaluation. Technical Report 1145, MIT AI Laboratory, July 1989.

[7] A. Bondorf, N. Jones, T. Mogensen, and P. Sestoft. Binding time analysis and the taming of self-application. Draft, DIKU, University of Copenhagen, Denmark, August 1988.

[8] D. Chase, M. Wegman, and F.K. Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, 1990.

[9] C. Consel. New insights into partial evaluation: the schism experiment. In *Proceedings of the 2nd European Symposium on Programming*, pages 236–246. Springer-Verlag, 1988. LNCS 300.

[10] C. Consel and O. Danvy. For a better support of static data flow. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture (LNCS 523)*, pages 496–519. Springer-Verlag, ACM, Cambridge, MA, August 1991.

[11] K.D. Cooper, M.W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 1993.

[12] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, New York, NY, 1977. ACM Press.

[13] R. Cytron and M. Burke. Interprocedural dependence analysis and parallelism. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 167–175, June 1986.

[14] W.J. Dally. 1993.

[15] A.J. Demers. Computation of aliases and support sets. In *Proceedings on the 1987 Symposium on the Principles of Programming Languages*, pages 274–283, 1987.

[16] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Proceedings on the 1990 Symposium on the Principles of Programming Languages*, pages 157–168, 1990.

[17] A.P. Ershov and V.E. Itkin. Correctness of mixed computation in algol-like programs. In J. Gruska, editor, *Mathematical Foundations of Computer Science, Tatranská Lomnica, Czechoslovakia. (Lecture Notes in Computer Science, Vol. 53)*, pages 59–77. Springer-Verlag, 1977.

[18] A.P. Ershov and N.D. Jones. Two characterizations of partial evaluation and mixed computation. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages xiii – xxi. North-Holland, 1988.

[19] F.E. Allen et. al. The experimental compiling system. *IBM Journal of Research and Development*, 24(6):695–715, November 1980.

[20] Y. Futamura. Partial evaluation of computation process – an approach a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[21] C. Ghezzi, D. Mandrioli, and A. Tecchio. Program simplification via symbolic interpretation. In S.N. Maheshwari, editor, *Foundations of Software Technology and Theoretical Computer Science. Fifth Conference, New Delhi, India. (Lecture Notes in Computer Science, Vol. 206)*, pages 116–128. Springer-Verlag, 1985.

[22] S.A. Gupta. Assessing the partial evaluation technology and its applicability to scientific computing. MIT EECS Area Examination Submission, 1991.

[23] M.A. Guzowksi. Towards developing a reflexive partial evaluator for an interesting subset of lisp. Master's thesis, Case Western Reserve University, Cleveland, Ohio, January 1988.

[24] M. Halfant and G.J. Sussman. Abstraction in numerical methods. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, 1988.

[25] A. Haraldsson. A program manipulation system based on partial evaluation. Linköping Studies in Science and Technology Dissertations 14, Linköping University, Sweden, 1977.

[26] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 28–40, 1989.

[27] P. Hudak. A semantic model of reference counting and its abstraction. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 45–62. Ellis Horwood Limited, Chichester, West Sussex, England, 1987.

[28] N.D. Jones. Challenging problems in partial evaluation and mixed computation. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 1–14. North-Holland, 1988.

[29] N.D. Jones and S.S. Muchnick. Flow analysis and optimization of lisp-like structures. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 102–131. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[30] K.M. Kahn. Partial evaluation, programming methodology, and artificial intelligence. *The AI Magazine*, 5(1):53–57, 1984.

[31] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[32] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[33] D. E. Knuth. An empirical study of fortran programs. *Software – Practice and Experience*, 1(2):105–133, 1971.

[34] M. Lam. Software pipelining: An effective scheduling technique for vliw machines. In *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation*, pages 318–328. ACM, June 1988.

[35] W. Landi and B. Ryder. Pointer-induced aliasing: A problem classification. In *Proceedings on the Eighteenth Annual Symposium on the Principles of Programming Languages*, pages 93–103, January 1991.

[36] W.A. Landi. *Interprocedural Aliasing in the Presence of Pointers.* PhD thesis, Rutgers University, January 1992.

[37] J. Launchbury. Projections for specialization. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation.*, pages 299–315. North-Holland, 1988. 625 pages.

[38] B. Liskov and J. Guttag. *Abstraction and Specification in Program development.* MIT Press, Cambridge, MA, 1986.

[39] U. Meyer. Techniques for partial evaluation of imperative languages. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, New Haven, Connecticut, 1991. ACM.

[40] B. Miller. The frequency of dynamic pointer references in "c" programs. *SIGPLAN Notices*, 23(6):152–156.

[41] T. Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Bjørner, P. Ershov A., and N.D. Jones, editors, *Partial Evaluation and Mixed Computation.*, pages 325–347. North-Holland, 1988. 625 pages.

[42] N. Osgood. Midas: An automatic system for the discovery and application of machine specific optimizations. BS Thesis, MIT EECS, June 1990.

[43] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C.* Cambridge University Press, second edition, 1992. 994 pages.

[44] S.E. Richardson. Evaluating interprocedural code optimization techniques. Technical Report CSL-TR-91-460, Computer Systems Laboratory, Stanford University, Stanford, CA, February, 1991.

[45] S.E. Richardson and M. Ganapathi. Interprocedural analysis useless for code optimization. Technical Report CSL-TR-87-342, Computer Systems Laboratory, Stanford University, Stanford, CA, November, 1987.

[46] E. Ruf and D. Weise. Using types to avoid redundant specialization. In *Proceedings of the 1991 Symposium on Partial Evaluation and Program Manipulation*, pages 321–333, New York, NY, 1991. Association for Computing Machinery.

[47] E. Ruf and D. Weise. Opportunities for online partial evaluation. Technical Report CSL-TR-92-516, Computer Systems Laboratory, Stanford University, Stanford, CA, April 1992.

[48] D. A. Schmidt. *Denotational Semantics: A Methodlogy for Language Development.* W. C. Brown, Dubuque, IA, 1988.

[49] R. Schooler. Partial evaluation as a means of language extensibility. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, August 1984.

[50] A. Srivastava and D.W. Wall. A practical system for intermodule code optimization at link-time. Technical report, Digital Western Research Laboratory, Palo Alto, CA, December 1992.

[51] B. Steensgaard. 1993.

[52] R.S. Wallace. *Numerical Iterative Hierarchical Clustering.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1989.

[53] M. Wegman and F. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[54] W. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Proceedings of the 7th Symposium on the Principles of Programming Languages*, pages 83–94, 1980.

[55] D. Weise. Graphs as an intermediate representation for partial evaluation. Technical Report CSL-TR-90-421, Computer Systems Laboratory, Stanford University, Stanford, CA, 1990.

[56] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture (LNCS 523)*, pages 165–191. Springer-Verlag, ACM, Cambridge, MA, August 1991.

[57] D. Weise and E. Ruf. Computing types during program specialization. Technical Report CSL-TR-90-441, Computer Systems Laboratory, Stanford University, Stanford, CA, May 1990.

[58] M. Yee. 1990.