FULCRA POINTER ANALYSIS FRAMEWORK

BY

ERIK MATTHEW NYSTROM

B.S., North Carolina State University, 1998
M.S., University of Illinois at Urbana-Champaign, 2002

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

ABSTRACT

Fulcra is a flexible and accurate pointer analysis framework. The algorithms developed permit accurate analysis of programs within SPEC and Mediabench benchmark suites generally within a small number of seconds. Even when a very detailed heap allocation model is used, the longest analysis time is 5 min, while similar frameworks require hours or months. Furthermore, comparisons of accuracy show that the algorithms developed for Fulcra can provide orders of magnitude difference between the least and most accurate analysis configurations.

While the implementation is exciting, the author's primary goal is not to laud its speed or accuracy, but to provide a guide to designers in how to construct very flexible pointer analysis frameworks. For example, the algorithms developed for Fulcra enable control of context sensitivity, field sensitivity, and heap sensitivity at the granularity of a single program object. It is only through this flexibility that frameworks can hope to find a true balance between accuracy and scalability capable of satisfying the timeliness demanded by some clients with the result quality demanded by others.

The contributions of Fulcra center around its whole-hearted embrace of hybrid pointer analysis algorithms and the design of elegant qualified constraint-based algorithms that enable simple, flexible, and accurate approachs to context and heap sensitivity.

To my wife for her unwavering love and support and to my bright-eyed son who would, at present, enthusiastically eat this text.

ACKNOWLEDGMENTS

My parents, brothers, and sisters all have my immeasurable gratitude for giving me a joyous childhood filled with curiosity and a fulfilling adulthood rooted in love and support.

I would like to thank my adviser Wen-mei Hwu for his guidance, support, and for creating an atmosphere where I was free to explore diverse computer architecture topics. Furthermore, my committee members Vikram Adve, Matthew Frank, Steve Lumetta, and Janak Patel have provided insightful feedback and guidance, for which I am grateful.

Over these many years, I had the opportunity to work closely with many gifted members of the IMPACT research group and was lucky to call them my friends. For this, I would like to thank John Sias for challenging my assumptions and expanding my vision, Hillery Hunter for providing new perspectives and guidance, Ronald Barnes and Matthew Merten for many years of fruitful research and helping my development as a researcher, Andrew Trick for showing dedication and grace under pressure, and Hong-Seok Kim for his unique view of theory.

Finally, I thank the National Science Foundation, Intel Corporation, DARPA/MARCO GSRC, the University of Illinois, and the Department of Electrical and Computer Engineering for their generous financial support which made my graduate studies possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Fulcrum - ful • crum; *pl.* ful • cra; *etymology* From Latin fulcire "to prop." The support about which a lever turns; One that supplies the capability for action.

Interprocedural pointer analysis has an extensive history in the research community. Though not revealing the difficulty of the problem, its goal can be simply stated as obtaining a description of the possible targets of a program's pointer based accesses. However, a vast number of complications and design issues erupt when one attempts to realize an implementation. It is likely this combination of simplicity and impossibility that has lured a decade of theoreticians and engineers.

A goal of this work is to document many of the impediments to efficient pointer analysis and to provide elegant and controllable mechanisms for overcoming them. This dissertation will hopefully serve as a guide to designers wishing to construct flexible pointer analysis frameworks.

The name *Fulcra* was chosen for two reasons. First, a fulcrum is, by itself, not much more significant than a door stop. Similarly, the potential of a pointer analysis system is wasted if not put into a position where some client can leverage its results. Such goals necessitate the engineering of a general pointer analysis framework capable of providing timely results. This effort brings with it many reality checks.

The second reason for the name choice is related to the fact that the constraint based analysis framework developed for Fulcra is very flexible (at least within the bounds of pointer analysis). For example, the degree of context sensitivity, field sensitivity, and heap sensitivity can all be varied at the granularity of a single program object. The analysis process uses this flexibility as a fulcrum for leverage toward a more efficient analysis process. Fulcra was chosen over fulcrum because the author preferred the shorter plural.

As its contributions, this dissertation provides:

1. The first framework to comprehensively embrace hybrid pointer analysis algorithms
2. A simple, flexible, and accurate approach to context sensitivity and heap sensitivity centered on elegant qualified constraint-based algorithms
3. One of the most efficient Andersen's style pointer analysis formulations
4. A configurable pointer analysis framework that supports in excess of 24 substantially different kinds of pointer analyses
5. A comparison across many different approaches to pointer analysis and the ramifications of different selections

The dissertation is organized into six major parts. Chapter 2 provides background on different elements of, and alternatives, in the design of pointer analysis systems. In addition to general background, it also motivates the selection of particular high-level designs from the available alternatives (such as an Andersen's style formulation).

Chapter 3 discusses the theoretical model for a context sensitive pointer analysis. The model facilitates the discussion of two critical concepts: (1) The end goal of context-sensitivity is to obtain pointer analysis results equivalent to those obtained had every path through the program's call graph been explicitly expanded and then analyzed context-insensitively. (2) No unrealizable data flow can occur if a program lacks procedural side effects. While providing a good starting point for understanding the challenges context sensitivity poses, the basic model has many shortcomings, making it an inappropriate choice for actual implementation.

Chapter 4 presents a novel approach to constructing a context-sensitive pointer analysis that uses qualified constraint-based algorithms that is both amenable to implementation and has many characteristics that enable more efficient analysis. In particular, maintaining context sensitivity is translated into a regular interaction among qualifiers, or constraint attributes, and permits an analysis to venture across a continuum spanning from context-insensitive to fully context-sensitive analyses. This is the primary mechanism necessary for cracking context-sensitive analysis, in particular when using Andersen's formulation.

Chapter 5 presents the approach used to obtain field sensitivity, in particular safety issues and the implications a field sensitive analysis has on a jointly context sensitive analysis. Chapter 6 presents a broad evaluation of Fulcra for context sensitivity, heap sensitivity, and field sensitivity. The evaluation includes comparisons of analysis time and result accuracy. Finally, Chapter 7 details opportunities for future research and development.

2. BACKGROUND AND SETUP

Conceptually, the goal of a pointer analysis system is to describe the potential targets of each pointer-based access. The memory locations that a program expression appears to access will likely influence the decisions made by a compiler or other software development tool. The heavy use of pointers by many applications makes pointer analysis useful, if not critical, for tasks like register promotion, scheduling, memory data flow, debugging, verification, as well as most program optimizations.

Pointer information could be obtained through the execution of a program for every possible input and the collection of all seen memory accesses. Similar in concept, and more likely to complete, pointer analyses use an abstract model of program execution to derive a conservative set of all possible pointer relationships. This chapter's goal is to provide background on many of the algorithmic options available to designers. Section 2.1 introduces important terms that will be used throughout this dissertation. Section 2.2 walks through the evaluation process used to gather empirical results. The remaining sections act as a pointer analysis primer, covering many facets of the pointer analysis problem and detailing many approaches that may be taken. They will also highlight the design choices made for this work. In addition to this chapter, we refer readers to [1, 2] for a more extensive set of references.

2.1 Terms

The following important terms will be used throughout this work:

1. Object - A collection of contiguous memory locations sharing a common name. Objects provide a static view of the the memory locations that a program may access at runtime. There is a one-to-one mapping between *named*, i.e., statically allocated, variables and an object, while a single object may represent many *anonymous*, i.e., dynamically allocated, memory locations.
2. Pointer Information - A relationship between program expressions (operations) based upon the objects these expressions (operations) may access.
3. Points-to - Object `a` points-to `b` if `a`'s value is the address of `b`.
4. Alias - Two expressions e_1 and e_2 alias if they resolve to the same object. For example, `*x` and `*y` alias if `x` and `y` reference the same object. While related, points-to and aliasing are different and knowledge of one does not necessarily translate into an equivalent form of the other.
5. Client - A consumer of pointer analysis results.
6. Realizable Result - A pointer analysis result that may occur for some execution of a program. Unrealizable will refer to all other results.
7. Optimistic Result - A pointer analysis result that is a subset of all realizable results.
8. Conservative Result - A pointer analysis result that is a superset of all realizable results. This may also be referred to as *Over approximation*.

9. Incorrect Result - The generation of an optimistic result for use by a client that requires conservative results to generate valid output. *Correct* or *Safe* will be used when a result is acceptable for particular clients.

10. Accuracy - A measure of how closely the derived pointer relationships match those realizable by the program. While some works use *precision*, this author prefers accuracy. Section 2.2 will cover this in more detail.

11. Scalability - A measure of an algorithm's ability to analyze a range of programs having a variety of characteristics without consuming an exorbitant amount of resources. Section 2.2 will cover this in more detail.

12. Procedural Side-effect - or side-effect, data flow whose consequences are visible to procedures outside the one containing the code that causes the data flow to occur.

13. Call Path - a sequence of procedure calls possible within the program's call graph.

There are many facets to the pointer analysis problem. The following outlines independent and complementary tactics to more faithfully model real program execution and thereby obtain more accurate results.

1. Assignment Modeling: *Andersen* style [3], also referred to as a *sub-typing* or *inclusion* based approach, and *Steensgaard* style [4], also referred to as a *unification* or *equivalence* based approach, represent the two extremes of modeling assignments. If a single location points to two different objects, Andersen will continue to track

the objects separately, while Steensgaard will unify them, tracking them as a single object. Steensgaard is a lossy compression of pointer information but is very efficient, almost linear in its pure form.

2. Context Sensitivity: A context-sensitive algorithm is able to keep data flow along different call paths separate. For example, should functions A and B both call C, a context-insensitive algorithm may show false data flow from A into B, by way of C. Approaches to context sensitivity are diverse and vary in the level of separation provided to interprocedural data flow, the modeling of recursion, and the treatment of indirect function calls.

3. Heap Sensitivity: Heap sensitivity enables an algorithm to distinguish between different heap allocated objects even though allocated by the same call to `malloc()`. Typically, the specialization of heap locations is strongly tied to context sensitivity, though other options exist [5].

4. Field Sensitivity: A field-sensitive algorithm is able to distinguish between fields of an aggregate object (a `struct` in the C language). In a field-insensitive analysis, the C-language expressions `x.f1` and `x.f2` are equivalent to `x`.

5. Flow Sensitivity: An algorithm is flow sensitive if its results account for the effects of control flow on the pointer data flow. The primary alternatives are full flow sensitivity [6, 7] and the use of static single assignment (*SSA*) [8] to obtain partial flow-sensitivity [9].

This dissertation focuses on analysis algorithms for an Andersen's style, context, field, and heap sensitive, but flow insensitive framework. The following sections provide additional background on and the reasoning behind these choices.

2.2 Empirical Evaluation

As with all studies, it is important to have a measure or measures of the quality of one result versus another. For pointer analysis, it is important to compare the cost of the analysis process and the potential usefulness of the result across a wide variety of applications. The following sections discuss the scalability (cost), accuracy (usefulness), and the applications used for evaluations.

2.2.1 Scalability

While most pointer analysis algorithms have a fairly high theoretical complexity (polynomial if not exponential in the worst case), the worst case may not be the common case and, for the same applications, there may be orders of magnitude difference in the real cost of two similarly positioned analyses.

This work uses a purely empirical measure of scalability and simply times the analysis process over a set of applications. While straightforward in concept, some previous works leave ambiguity as to what is being timed or state that some processing is left for demand driven queries. All timing measurements in this dissertation start once the program expressions are loaded (an abstract syntax tree (AST) in our case) and conclude once the complete and concrete analysis results have been produced. The measured analysis times do not include the annotation of these results back to the AST, but the results are in a form where they do not require any additional refinement. Thus, the timing includes all algorithmic iteration, call graph discovery, simplifications, and pointer derivations. Unless otherwise stated, all timing results will be presented in seconds.

Keep in mind that application size is not always a good indicator of the resultant cost of the analysis process. For example, a 10 million line program lacking pointer use should be easy to analyze. There are many programs around 50 000 lines of code (*LOC*) that vary greatly in difficulty and, in the end, analysis time.

2.2.2 Accuracy

Measuring the usefulness of the results of a particular pointer analysis is surrounded by much disagreement. The options can be roughly divided into three categories: *raw*, *extrapolated*, and *client based*.

A raw measure is some metric based purely on the analysis process. The main metric used is a count of points-to relations. As long as an analysis framework computes the final, concrete points-to relations (which is not a given since some frameworks perform this step on a demand driven or offline basis), these results are straightforward to obtain. However, they are also the most difficult to interpret. While a reduction in the number of points-to relations is desirable, the relationship between points-to and data flow is ambiguous. The biggest drawback is that a raw points-to count loses its meaning if two

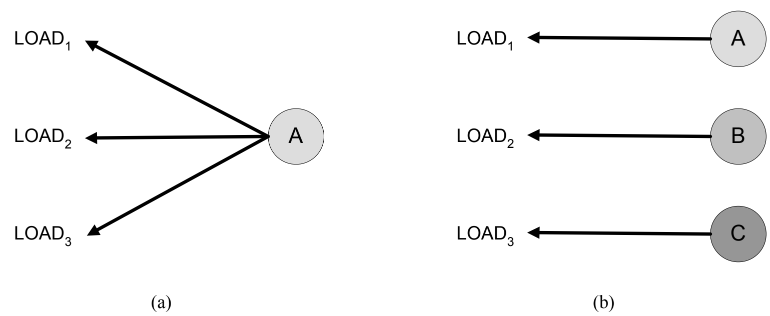


Figure 2.1 Two different pointer analysis results where (a) three loads access the same object and (b) three loads access three unique objects. In calculating a measure of accuracy, the raw points-to size is 3 for both (a) and (b), while the measure of accesses per object is 3 for (a) and 1 for (b).

analyses result in different numbers of objects. For example, a heap sensitive analysis may specialize heap locations to obtain better results but the increase in objects will likely inflate the number of points-to relations even though the results may be better. To demonstrate this, Figure 2.1 shows analysis results where all the loads in (a) all access the same object and the loads in (b) access distinct objects. The points-to count for both is 3 even though they are clearly different results.

The most concrete measure of accuracy is the effect some result has on a client. First, this requires the existence of a client that can leverage the results which creates a chicken-and-egg dependence. Without first obtaining more accurate results it is unlikely that someone will have developed a client that can truly leverage the benefits. A related problem is that benefit to one client is not necessarily indicative of benefit to another. The most complex issue, however, is that the most interesting clients are themselves complex beasts and may not have a well behaved response to variations in analysis results. For example, a scheduling client may, due to approximations in its algorithms, provide a better result when guided by less accurate pointer analysis result.

Extrapolated measurements lie between raw and client based ones. In a sense, they are a primitive, well-behaved clients. One example of an extrapolated measurements is pair-wise dependences (*PWDep*), a measure of the number of dependences found when some set of program expressions are compared. However, PWDep has a few nonobvious drawbacks, specifically:

1. Nonlinearity - The term “pair-wise” means that some set of expressions are used to perform a n^2 set of comparisons. Because of this nonlinearity the scope of the chosen expression set heavily influences the result calculated, diluting differences, and may mislead researchers trying to interpret the results. For example, if 10 expressions are selected for comparison and, for one set of results, 6 out of 45 comparisons are dependent and, for another, 5 out of the 45 are dependent the PWDep is 13.3% and 11.1%, respectively, an absolute difference of about 2%. Had five expressions, dependent on each other but independent of the other 10, been included into the calculation, the PWDep would become 16 out of 105 (15.2%) and 15 out of 105 (14.2%), an absolute difference of only 1%. The small expansion in set size substantially diluted the change in PWDep. A good example from the literature is Das et al. [10]. This work uses a measure termed Alias Frequency which is identical to PWDep except that it includes read-read “dependences” (any two expressions that alias). The lower bound alias frequency computed over the

scope of an entire function for the benchmark *132.ijpeg* is about 6% while Das's algorithm provides 18% and Steensgaard's 19%. Their conclusion is essentially that 12% is fairly close (or 5% when averaged across their benchmark set). At best it is difficult to interpret the meaning of 12%. However, over a function scope it is likely due to a large change in the results. Consider that, given 200 expressions, if none alias the alias frequency is 0% and if 64 completely alias the frequency is 10% (2016 aliases out of 19 900 comparisons).

2. Client Specificity - While not a true client, PWDep assumes that one cares only about dependences. Two expressions that appear to read a common datum are independent. A more accurate analysis process may determine that the expressions, in fact, read from different locations. However, this will have no impact on the PWDep even though a client that helps with debugging or attempts to layout data for cache locality may benefit from this determination. Figure 2.1 gives an excellent example of this, where the PWDep for both (a) and (b) is zero even though the results are quite different. (This is the primary benefit of alias frequency over PWDep as it does not suffer from this issue.)

This dissertation relies upon an extrapolated measure called accesses-per-object, *ApO*. ApO has similarities both to raw points-to relations and to pair-wise dependences. ApO is calculated by counting the number of expressions that appear to access (load from or store to) a particular program object. This count can be used directly or averaged across the number of objects. Unlike PWDep, ApO relies upon a simple mapping from objects

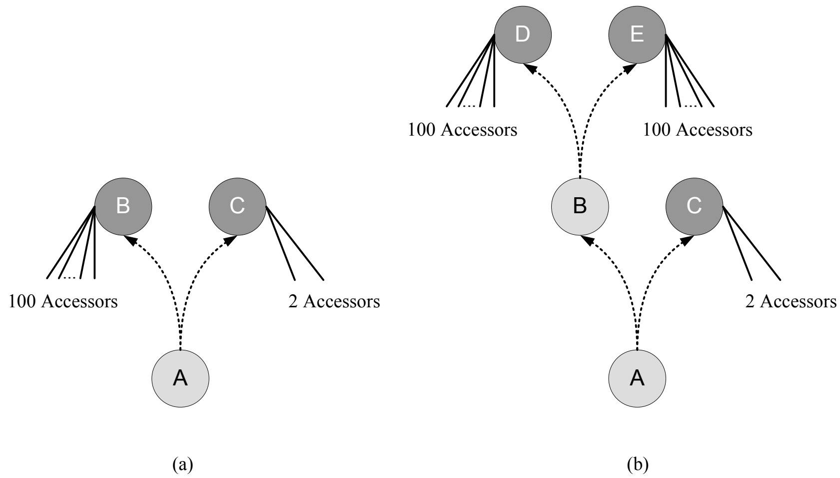


Figure 2.2 Example used to show how to calculate accesses per object in the presence of differing amounts of heap specialization. In (a), in a successful attempt at obtaining a better result, object `A` has been specialized into two objects `B`, with 100 accessors, and `C`, with only two accessors. In (b), `B` has been specialized without benefit into `D` and `E`, each still having 100 accessors.

to accessors and does not suffer from nonlinearity. Since it treats load and store accesses equally, it is less client specific. In comparison to raw points-to, ApO can be calculated in a way that permits comparisons between results in which the number of modeled heap objects differs.

Figure 2.2 depicts two possible analysis outcomes and differs only in how much heap specialization was performed. In (a), object `A` has been specialized into two objects `B`, with 100 access-or, and `C`, with only two accessors. The specialization of `A` has shown benefit because `B` and `C` diverge in how they appear to be accessed. In (b), `B` is further specialized into `D` and `E`, each having 100 accessors. This specialization was not beneficial because `D` and `E` are still identical. Since the specialization attempt for (b) provides no benefit, the results are essentially equivalent. The measure of accuracy used should reflect this equivalence.

If calculated naïvely, the ApO for (a) would be *average-of*(100,2), or 51, and for (b) it would be *average-of*(100,100,2), or 67. Though equivalent, the unbalanced cloning of a heavily accessed object has skewed the average upward making (b) appear worse than (a). For specialized objects ApO should be calculated as a series of averages where a parent's value is the average of its children. Using this approach, the ApO for (a) would still be *average-of*(100,2). However, for (b), a value for `B` would be computed first as *average-of*(100,100) and this value averaged with the value for `C`. The final value would be *average-of*(*average-of*(100,100),2), or 52, identical to that obtained for (a).

2.2.3 Benchmarks

Empirical evaluations throughout this dissertation are benchmarked against the applications listed in Table 2.1. Table 2.1 lists all (nonrepeating) C benchmarks from SPECint92, SPECint95, and SPECint2000, the largest benchmarks from Mediabench, and three independent applications. It includes a description and some basic program characteristics. Many results will be shown across the entire suite. However, for brevity, a subset will be used when it is sufficient to make the desired point.

2.3 Andersen' Style Assignment Modeling

The decision to design an Andersen's style (as opposed to a Steensgaard's style) analysis impacts the design of many pointer analysis algorithms as well as the resultant

Table 2.1 Benchmarks used for empirical evaluation, including the abbreviation used in graphs, the source benchmark suite, lines of code, number of indirect call sites, and a brief description. *The indirect function call in 197.parser is due to the stdlib procedure `qsort()`.

Benchmark	Abbrv	Suite	LOC (K)	Ind CS	Description
008.espresso	008	SPECint92	13	15	PLA logic optimization
023.eqntott	023	SPECint92	20	11	Bool eqn to truth table
099.go	099	SPECint95	45	0	"Go" game player
124.m88ksim	124	SPECint95	33	3	Motorola 88k simulator
129.compress	129	SPECint95	18	0	Compression
130.li	130	SPECint95	23	4	Lisp interpretor
132.ijpeg	132	SPECint95	26	644	JPEG photo decoder
134.perl	134	SPECint95	40	12	Perl interpretor
164.gzip	164	SPECint2000	24	2	Compression
175.vpr	175	SPECint2000	33	2	FPGA place and route
176.gcc	176	SPECint2000	222	141	GNU compiler
181.mcf	181	SPECint2000	18	0	Mass-transit vehicle scheduler
186.crafty	186	SPECint2000	35	0	Chess game player
197.parser	197	SPECint2000	27	1*	English syntactic parser
253.perlbmk	253	SPECint2000	74	58	Perl interpretor
254.gap	254	SPECint2000	76	1281	Group theory
255.vortex	255	SPECint2000	69	15	Obj-oriented database
256.bzip2	256	SPECint2000	21	0	Compression
300.twolf	300	SPECint2000	36	0	Place and route
gsmdec	gsmd	Mediabench	22	6	Sound decompression
gsmenc	gsme	Mediabench	22	6	Sound compression
jpegdec	jpgd	Mediabench	24	600	JPEG photo decoder
jpegenc	jpge	Mediabench	23	654	JPEG photo encoder
h263dec	263d	Mediabench	22	0	H.263 video decoder
h263enc	263e	Mediabench	23	0	H.263 video encoder
mpeg2dec	mg2d	Mediabench	25	0	MPEG-2 video decoder
mpeg2enc	mg2e	Mediabench	23	0	MPEG-2 video encoder
mpg123	mpl3	Independent	24	33	MPEG-2 layer 3 audio decoder
jpeg2kdec	jg2k	Independent	43	27	JPEG-2000 part-1 standard
mpeg4dec	mpg4	Independent	66	0	MPEG-4 video decoder

efficiency and accuracy of the pointer analysis framework. The importance of this decision warrants both motivation and explanation.

Research publications, such as [11–13], have shown that unification based pointer analyses are very scalable. Object merging necessary to restrict points-to set size to one provides a strong bound on the quantity of analysis information as well as performing a form of preemptive redundancy removal. However, the same restrictions that guarantee scalability can also lead to unrealizable data flow and, in the end, less accurate results.

Inclusion based analyses maintain appropriate subset relationships due to assignments and therefore preserved the expected data flow between objects. For example, given the assignments `a:=b` and `a:=c`, an inclusion based analysis explicitly preserves the facts that `points-to(a)` $\supseteq$ `points-to(b)` and `points-to(a)` $\supseteq$ `points-to(c)`. The following will walk through the primary reasons why an inclusion based analysis was chosen.

First, an inclusion based pointer analysis can provide more accurate and intuitive results. The typical justification is that a more faithful model of program data flow may result in fewer dependences and an immediate increase in software performance. While possible, this statement assumes that existing tools can take advantage of better information and overlooks benefits like providing a more realistic view of data flow to program understanding and debugging tools and reducing the time to perform analyses and transformations.

Consider the excerpt from *jpegdec* shown in Figure 2.3. The common use of variable `dest` will result in a unification based analysis merging the five objects assigned into

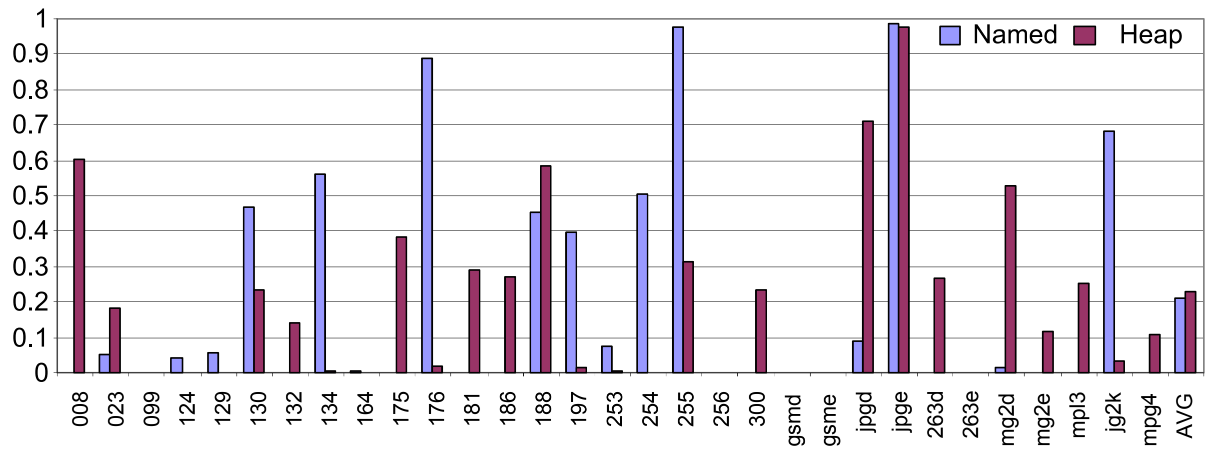
```
1:  switch(format) {
2:      case BMP: dest = jinit_bmp(&cinfo); break;
3:      case GIF: dest = jinit_gif(&cinfo); break;
4:      case PPM: dest = jinit_ppm(&cinfo); break;
5:      case RLE: dest = jinit_rle(&cinfo); break;
6:      case TARGA: dest = jinit_targa(&cinfo); break;
7:  }
8:  (*dest->start_output)(&cinfo, dest);
9:  (*dest->start_output)(&cinfo, dest);
```

Figure 2.3 Excerpt from *jpegdec* showing code that can benefit from an Andersen's style analysis.

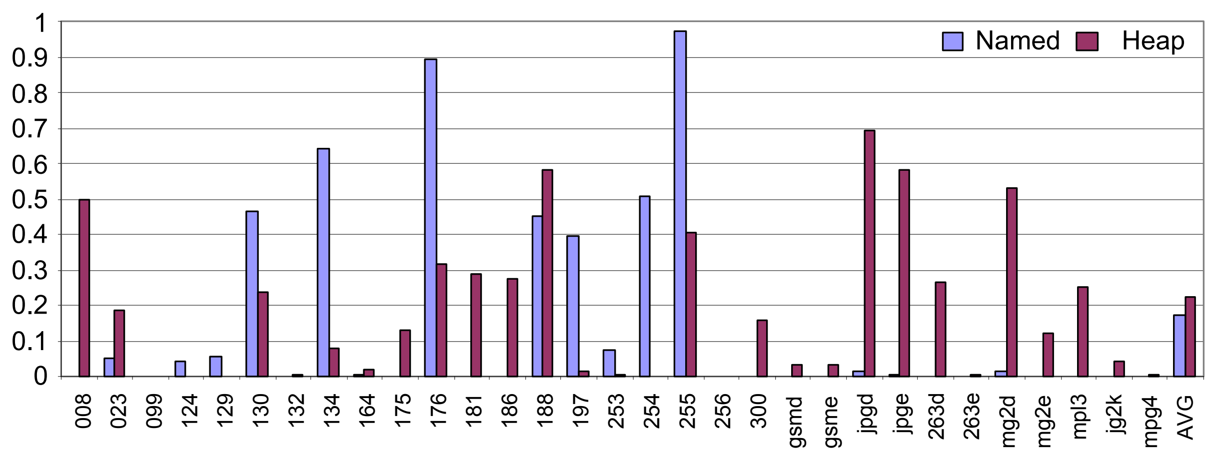
`dest` at lines 2-6. However, the objects created by the `jinit_XXX()` procedures contain very different pointer information and unrealizable data flow created as this merging cascades to other objects. An inclusion based analysis will preserve these distinct objects. Furthermore, a technique described in Section 7 can be used to further refine these results by leveraging the correlation between the datum controlling the indirect function call `*dest` and the parameter `dest`.

A broad comparison of accuracy is shown in Figure 2.4. Figure 2.4(a) and (b) show the reduction in accesses-per-object observed if an Andersen's style analysis is used instead of a best-case formulation of Steensgaard's. Due to a design constraint of Fulcra, the analysis used a Steensgaard's bottom-up process but an Andersen's style top-down phase. Figure 2.4(a) models the heap per-call-site while (b) uses a more accurate model.

Over half of the benchmarks show a substantial improvement in accuracy for an Andersen's style analysis versus the Steensgaard's style even though the Steensgaard data was able to leverage field, context, and heap sensitivity to keep objects separate.



(a)



(b)

Figure 2.4 Comparison of accuracy between a Steensgaard's and Andersen's formulation for a context, field, and heap sensitive analysis. Graphs (a) and (b) display a lower-bound on the percentage reduction in accesses-per-object obtained by using an Andersen's formulation where (a) models the heap per-call-site and (b) uses a more accurate heap model. The data is a lower bound because the experiment used a Steensgaard's bottom-up process but, due to a design constraint of Fulcra, an Andersen's top-down.

Another reason for the selection of a inclusion-based algorithm is the hope that the resultant accuracy will be more resilient to imperfections in the analysis process. No pointer analysis is perfect, and unrealizable data flow will always exist. Even though a program's true data flow should not result in the object merging, a pointer analysis's imperfect abstract view of data flow may force it to do so.

The final reason is that the costs and benefits are largely unproven. This final point, which could be labeled academic curiosity, provides valuable upper-bounds on accuracy to help guide the construction of true hybrid analyses. Along these lines, very few frameworks have combined similar degrees of accuracy and, of these, Fulcra is many times faster and far more configurable, thus allowing for greater design space exploration.

There are domains in which a unification-based analysis will be a better choice. However, the trade-offs are not well understood. Furthermore, as this dissertation shows, Fulcra is often extremely fast (milliseconds to a few seconds) for many nontrivial applications. If similar applications are of interest to a developer, the use of this formulation is worry-free.

2.4 Basics of Context Sensitivity

Modern programming practices encourage code reuse, which often results in programs composed of a complex network of procedure calls. For static analysis, this exacerbates the problem of unrealizable interprocedural data flow [14]. *Context-sensitive* analyses are often able to avoid unrealizable data flow due to procedure calls, thereby delivering a higher degree of accuracy than their context-insensitive counterparts.

In the literature, there have been many approaches to context-sensitive pointer analysis. Some analyses [6, 15–18] mimic dynamic execution, repeating call-return sequences until the analysis reaches a global fixed point. While some amount of memoization is possible, in essence a procedure must be reanalyzed for every call path to which it belongs, leading to serious scalability issues. More recent work [7, 13, 19–21] formulates pointer analysis as a two-phase computation. This formulation has the advantage that it can be designed to analyze a procedure at most twice when given a fixed acyclic call graph. This dissertation takes a two-phased approach. Chapters 3 and 4 provide more details.

Regardless of the mechanism, the end goal of a context sensitive analysis is to obtain pointer analysis results equivalent to that obtained if a program's call graph were explicitly expanded, analyzed, and the results applied back to the original program. Full expansion converts data flow along call paths through shared procedures into explicitly separate data flow. Consider the example code shown in Figure 2.5(a). A context insensitive analysis is incapable of separating the data flow between `Yank()` and `Zulu()` from that between `Xray()` and `Zulu()`. This leads to the unrealizable results `a:=2` and `b:=1`. The code in Figure 2.5(b) is obtained by specializing the two calls to `Zulu()`. No procedures are shared and a simple analysis of program yields only the desired, context sensitive results `a:=1` and `b:=2`. The application of the results back to the original

```
Yank() {
    Zulu(&a,1)
}

Xray() {
    Zulu(&b,2)
}

Zulu(p,q)
    *p := q
}
```

(a)

```
Yank() {
    Zulu1(&a,1)
}

Xray() {
    Zulu2(&b,2)
}

Zulu1(p1,q1)                 Zulu2(p2,q2)
    *p1 := q1                    *p2 := q2
}                            }
```

(b)

Figure 2.5 The code in (a) can benefit from a context sensitive analysis and (b) shows how complete call graph expansion explicitly separates interprocedural data flow thereby obtaining context sensitivity. This provides a conceptual baseline for a context sensitive pointer analysis.

program involves only making $\texttt{p:=p}_1 \cup \texttt{p}_2$ and $\texttt{q:=q}_1 \cup \texttt{q}_2$. Otherwise, the results are unchanged.

Provided this baseline, context sensitivity can be divided into two pieces termed bottom-up and top-down context sensitivity. Bottom-up context sensitivity provides a procedure F a result equivalent to the full expansion of all call paths rooted at F. Top-down context sensitivity provides a procedure F a result equivalent to the full expansion of all call paths terminating at F. Both are necessary to obtain a truly context-sensitive result. The publications [22, 23] and this dissertation provide the first efficient application of bottom-up and top-down context sensitivity for C programs.

2.4.1 Bottom-up context sensitivity

Bottom-up context sensitivity can be obtained by visiting procedures in a reverse-topological order, forming compact models, or *summaries*, for each procedure and replacing all calls to the procedure with a specialized copy of its summary. The use of summaries is analogous to expanding the call graph in that interprocedural data flow is specialized. However, summaries may be far more compact than the replication of entire procedures and enable a more efficient implementation. The constructed summaries can be either *context specific* or *context independent.*

A context specific summary is constructed for a particular alias relationship among a procedure's inputs (parameters or global variables). Summarizies for a particular calling context can be formed as a simple relation in terms of only the procedure's inputs. However, the summaries are not globally valid and can only be used for call sites in which the inputs have matching alias relationships. New summaries must be constructed for each kind of input encountered. Algorithms that mimic dynamic execution, such as [6], form context specific summaries.

Context independent summaries can be applied universally and are as accurate as context specific summaries. The use of context independent summaries means that a procedure need only be summarized once. The original procedure is a trivial example of a context independent summary. Unlike context specific summaries, *it is not always possible to construct context independent summaries in terms of only a procedure's inputs.* This is an extremely important point and one that is often overlooked. Figure 2.6(a)

```
whiskey(w,x,y,z) {
   *x := &a;
   **y := z;
   *a := w;
}
```

(a)

if `*x` and `*y` don't alias

```
**y := z
```

(b)

if `*x` and `*y` alias

```
**y := z
*z := w
```

(c)

Figure 2.6 (a) Code example for which a context independent summary cannot be formed in terms of only function inputs, (b) side-effects if `*x` and `*y` do not alias, and (c) if `*x` and `*y` alias, `**y := z` and `*x := &a` combine to for the data flow `a := z` and the subsequent side effect `*z := w`.

shows a procedure `whiskey` whose context independent summary must include terms other than simple expressions of its inputs. Procedure `whiskey()`'s side-effects are dependent on whether or not `*x` and `*y` alias, shown in Figure 2.6(b) if they do not and and Figure 2.6(c) if they do. The context independent summary for `whiskey()` cannot be reduced from the original procedure. The algorithms in this dissertation form context independent summaries.

2.4.2 Top-down context sensitivity

At its essence, top-down context sensitivity provides a result identical to the individual application of each of a procedure's calling contexts. This can be achieved by fully expanding all call paths leading to and including the procedure, but, as with any call graph expansion, this can become extremely expensive.

Because top-down context sensitivity has had little coverage in literature, Figure 2.7 presents a code example, two failed approaches to top-down context sensitivity, and one

```
Victor() {
    Uni₁(&a,1,&a);
    Uni₂(&b,2,&a);
}

Uni(p,q,r) {
    *p := q;
    y := *r;
}
```

(a)

Calling Context

```
a:=1, b:=2
p:=&a,&b
q:=1,2
r:=&a
```

```
Uni(p,q,r) {
    *p := q;
    y := *r;
}
```

Analysis Result

```
a:=1,2
b:=1,2
y:=1,2
```

(b)

Calling Context

```
a':=1, b':=2
p:=&a',&b'
q:=1',2'
r:=&a'
```

```
Uni(p,q,r) {
    *p := q;
    y := *r;
}
```

Analysis Result

```
a':=1',2'
b':=1',2'
y:=1',2'
```

(c)

Calling Context

```
a:=1, b:=2
p:=&a,&b
q:=1,2
r:=&a
```

```
Uni(p,q,r) {
    y := *r;
}
```

Analysis Result

```
a:=1
b:=2
y:=1
```

(d)

Figure 2.7 Example explaining top-down context sensitivity (a) code example, (b) an approach leading to a loss of context sensitivity, (c) an approach maintaining only bottom-up context sensitivity, and (d) a truly top-down context sensitive result.

valid approach that does not require full call graph expansion. Figure 2.7(a) shows a code example in which `Victor()` calls `Uni()` twice. Similar to the example in Figure 2.5, the procedure calls and the first two parameters result in `a:=1` and `b:=2`. A third parameter has been added which is `&a` for both calls to `Uni()` and is read but not modified by `Uni()`.

Figure 2.7(b)-(d) show only the top-down step for `Uni()` to the analysis process and assume that a context sensitive result has already been derived for `Victor()`. The bracket assignments above `Uni()` is its calling contexts and the bracket assignments to the right of `Uni()` are the results of the particular top-down attempt.

Figure 2.7(b) naïvely applies `Uni()`'s calling context and derives both unrealizable results for `a` and `b` leading to the result that `y` could be either `1` or `2`. This approach resulted in a context-insensitive solution for both `Victor()` and `Uni()`.

Figure 2.7(c) depicts a more common tactic to the top-down pointer analysis phase. Instead of sharing an object like `a` between the procedures, Figure 2.7(c) provides `Uni()` a private copy of the calling context. While `y` obtains the same unrealizable result, `Victor()`'s `a` and `b` are isolated and remain accurate. The end result is only bottom-up context sensitive.

Figure 2.7(d) shows an approach, which will be covered in detail in Chapter 3, that shares objects as in Figure 2.7(b) but deletes the side-effect causing assignment `*p:=q` before applying the calling context. This preserves the results derived in `Victor()` while obtaining only the realizable result in `Uni()`. The result is fully context sensitive without requiring individual application of each calling context (i.e., full expansion of the call graph). The approach in Figure 2.7(d) is similar to the process advocated by this dissertation.

A final comment is that top-down context sensitivity is incompatible with unification based analyses. The goal of top-down context sensitivity is to provide an analysis result equivalent to one where the procedure's inputs where considered individually. However, a unification based analysis cannot represent the values of a particular input separately. Given a setup similar to that shown in Figure 2.7(d), a unification based analysis will merge `a` with `b` and `1` with `2`. Since `1` and `2` can no longer be differentiated, `a` will appear to contain either, as if `Victor()` has been analyzed context insensitively, and `y` will appear to obtain both.

2.4.3 Recursive procedure calls

The baseline used for a context sensitive analysis has been explained as a fully expanded call graph. However, in the presence of recursion complete call graph expansion is impossible. In such cases, recursive interprocedural data flow is treated context-insensitively.

Recursion is found by detecting SCCs in the call graph. All procedures within an SCC are merged into a common metaprocedure. The metaprocedure maintains distinct formal parameters for each member procedure and distinct call sites to procedures outside the SCC. However, all call sites to members of the SCC are converted to direct copy assignments between corresponding actual and formal parameters.

With call graph cycles collapsed, the call graph is left acyclic. It is against this acyclic call graph that the baseline for context sensitivity is derived. However, it is possible to treat recursion context-sensitively, or at least partially so. This has been done for data flow analysis of languages lacking aliasing [24] and an approach to this for a formulation based on this dissertation is shown in Chapter 7.

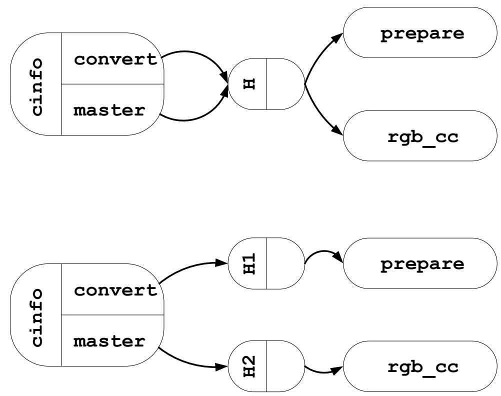
2.5 Heap Sensitivity

Many programs allocated objects on the heap through call to routines similar to `malloc()`. Many important and distinct data structures may be allocated from a common static call to an allocation routine. Heap sensitivity in pointer analysis attempts to separate distinct heap objects by modeling specialized instances of heap objects.

Figure 2.8(a) is a small example derived from a JPEG photo decoder. For this example, fields `colorc` and `prepare` are at the same positions within their respective data structures. Procedure `alloc()` is a custom wrapper around the standard allocation routine `malloc()`. If executed, two heap objects would be allocated, one created in `color_deconverter()` and the other in `master_decompress()`. While the indirect function call within `main()` can only call `prepare()`, it will appear to call `rgb_cc()` as well, if the pointer analysis does not differentiate the heap object for `cinfo->convert` from the

```
main() {
    color_deconverter(cinfo);
    master_decompress(cinfo);
    (*cinfo->master->prepare)(cinfo);
}
color_deconverter (cinfo) {
    cinfo->convert = alloc();
    cinfo->convert->colorc = rgb_cc;
}
master_decompress (cinfo) {
    cinfo->master = alloc();
    cinfo->master->prepare = prepare;
}
alloc() {
    return malloc();
}
```

(a)



(b)

Figure 2.8 Example showing the benefit of specializing heap objects. (a) Example based on a JPEG photo decoder/encoder, and (b) two models of the heap where the upper figure performed no specialization and the lower created two specialized objects.

one for `cinfo->master`. Figure 2.8(b) shows the two scenarios where the upper figure models a shared heap object and the lower two special heap objects.

Heap specialization is performed during the bottom-up phase of the two-phase computation where call chains are walked backward from callees to callers. It begins by assigning a unique heap object for each allocation routine. As the procedure summary of a callee is inserted into a caller's call site, a fresh (specialized) heap object is introduced for each heap object in the summary.

The specialization process can become expensive if performed needlessly because a heap object may be created for every call path terminating in an allocation routine. The following works discuss reducing this overhead [23, 25] and Section 4.7 covers additional details of the heap specialization.

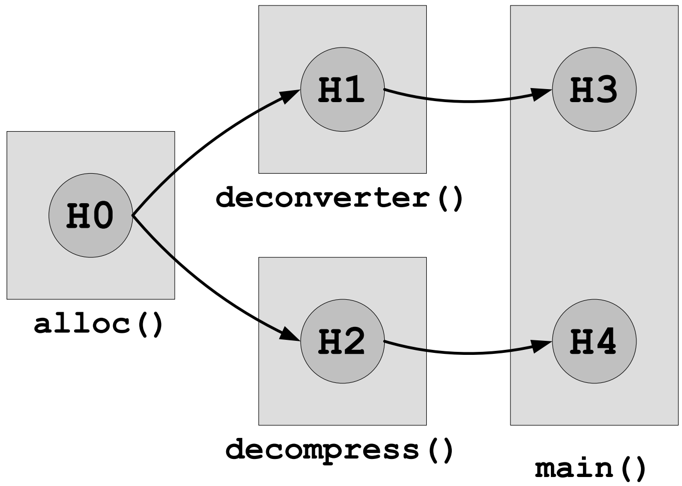


Figure 2.9 Graphical representation of a the lineage for specialized heap objects from the code example in Figure 2.8(a).

Future discussions will use a concept of heap lineage. Heap lineage is parent-child relationship between heap objects created during the bottom-up specialization process. Referring back to Figure 2.8(a), the heap object modeled for the analysis of procedure `alloc()` is the parent of the two objects modeled for the analysis of procedures `master_decompress()` and `color_deconverter()`. For the example, five heap objects may actually be modeled. Figure 2.9 shows a graphical representation of their lineage.

2.6 Indirect Function Calls

The pointer analysis discussions thus far have assumed the existence of a program call graph. Minimally, a call graph is necessary to know which program expressions to analyze. However its importance is elevated when attempting to perform a context sensitive analysis.

In the absence of indirect function calls, construction of a call graph entails a simple and transitive traversal of function calls, starting from the top-level procedure (for example, `main()`). Indirect function calls complicate the process because a pointer variable now controls call targets, making call graph construction dependent upon the pointer analysis that uses it. This creates a cyclic dependence between pointer analysis and call graph construction.

There are conservative and optimistic approaches to managing this cyclic dependence. A conservative approach frees call graph construction from pointer analysis by forming an over approximation of the call graph. This over approximation can be obtained by allowing each indirect function call to call all possible function targets, by using prototype information to prune targets, or by performing some other analysis [5, 26]. Using all possible targets is likely very inaccurate, while pruning using function prototypes is inaccurate at best and likely incorrect.

Optimistic approaches embrace the cyclic relationship by iterating between call graph construction and pointer analysis. The process begins by constructing a call graph composed of only direct calls. Pointer analysis is performed, and, if new indirect function targets are discovered, another round of call graph construction and pointer analysis is performed.

When used within a context sensitive environment, optimistic approaches can, themselves, be subdivided into those that perform *call site hoisting* [13] and those that perform *summary updating* [19, 20, 22]. The example in Figure 2.10(a) will be used to demonstrate

```
tango() {
    u:=*v;
    sierra(u,q);
}
sierra(s,q) {
    r:=*s;
    romeo(r,q);
}
romeo(p,q) {
    o:=*p;
    (*x)(o,q)
}
```

(a)

```
tango() {
    u:=*v;
    r:=*u;
    o:=*r;
    (*x)(o,q);
}
sierra(s,q) {
    r:=*s;
    o:=*r;
    (*x)(o,q);
}
romeo(p,q) {
    o:=*p;
    (*x)(o,q)
}
```

(b)

```
tango() {
    u:=*v;
}
sierra(s,q) {
    r:=*s;
}
romeo(p,q) {
    o:=*p;
    (*x)(o,q)
}
```

(c)

Figure 2.10 Code example (a) is used to demonstrate the difference between (b) call site hoisting and (c) summary update approaches to optimistic call graph construction within a context sensitive analysis.

the pros and cons of each approach. During the bottom-up process call site hoisting forms summaries that embody externally visible indirect call sites. Just as with side-effects, the summaries hoist specialized copies of indirect function calls into callers. This is the most accurate mechanism for handing indirect function calls within a context sensitive environment because call targets can be context specific. However, as shown in Figure 2.10(b), it can also be very expensive because summaries must account for the worst-case side-effects of a call target. This can lead to unnecessarily large summaries since the real targets may have few side-effects.

Summary updating speculates that a call target has no side effects and, if shown to be incorrect, must update any generated summaries to include newly discovered side-effects. Figure 2.10(c) shows an update based bottom-up process where the indirect procedure call does not have a side-effect. In effect, call site hoisting takes the worst case cost up front while summary updating assumes the best case and expands from there.

This dissertation uses call graph updating because, in general, most procedures have few if any side effects and what effects they have are largely localized within the call graph. Section 4.4 presents a technique that reduces the number of iterations, and therefore updates, by accelerating the call graph discovery process.

2.7 Field Sensitivity

Many C programs use aggregate data structures composed of multiple subcomponents or fields. A pointer analysis framework can choose to ignore field accesses, i.e., perform a field insensitive analysis. Field information can be used to provide better resolution into the use of aggregate data structures. For example, given a structure `st` it may isolate accesses to `st.f1` from those to `st.f2`. There are a large variety of mechanisms through which some form of field sensitivity can be achieved.

Approaches to field sensitivity can be roughly divided into those that use field names and indices, thereby achieving machine independent analysis results [27–29], and those that use field offsets, thereby relying upon a concrete mapping of data structures to memory [6, 19, 30]. Orthogonal to this categorization, approaches can also be partitioned

```
{
    struct {
       int *f1, *f2;
       char g1,g2,g3,g4;
       int *f4 } a;
    struct {
       int *f1, *f2;
       int *f3 } b;
    u := &a;
    u := &b;
    a.f1 := &x;
    a.f4 := &y;
    b.f1 := &z;
    b.f3 := &w;
}
```

Insensitive	**Steensgaard**
pt(a)={x,y};	pt(a.f1)={x,z};
pt(b)={z,w};	pt(a.f4)={y,w};
	pt(b.f1)={x,z};
	pt(b.f3)={y,w};

Based	**Indexed**	**Offset**
pt(off.0)={x,z};	pt(a.0)={z};	pt(a.0)={x};
pt(off.8)={w};	pt(a.6)={y};	pt(a.12)={y};
pt(off.12)={y};	pt(b.0)={z};	pt(b.0)={z};
	pt(b.2)={w};	pt(b.8)={w};

Figure 2.11 A sample program and the points-to results that can be obtained for five approaches to field sensitivity: Insensitive, Steensgaard, Based, Indexed, and Offset.

into those that maintain only a single global aggregate object [30], those that begin from from assembly [31], those that target a language other than C (such as Java [26, 32, 33]) or a more well behaved subset of C, and those that provide a mixture of static and dynamic type checking [34].

Figure 2.11 shows sample code sequence and the points results obtained by five different styles of field sensitivity. *Insensitive* is a field insensitive analysis, *Steensgaard* is a field name based analysis proposed in [28], *Based* is a field only model from [30], *Indexed* is a index based approach from [27], and *Offset* an offset based approach. Note that, for this example, *Indexed* and *Offset* yield the same results though for different reasons.

Use of field names to provide field sensitivity provides results independent of a particular target machine. If field names are used in a naïve way (essentially assuming a type safe language), false data flow may appear through similarly named fields and real data may be missing along intended paths. Yong et al. [29] and Steensgaard [28] take similar and robust approaches to this problem. If data flow is determined to exist between two data structures, the flow is field sensitive for the compatible prefix of the data structures while subsequent, incompatible fields are collapsed. Yong et al. provide a broad coverage of issues and include an offset-based model for comparison.

Pearce et al. [27] present a field index based approach. Instead of explicit field names or physical offsets, they number the fields and use these indices to match data flow. While at a vulnerable point between field name based and offset based approaches, the work provides an excellent discussion of the existence of cycles of pointer arithmetic and provides a faithful modeling of the arrow operator (as implicit pointer arithmetic).

While machine dependent, offset based approaches provide a more exact model of possible data flow. The offset-based approaches by Wilson and Lam [6] and Cheng and Hwu [19] have goals most similar to that presented in this dissertation. Wilson's framework avoids casting issues through complete reliance on field offsets and includes a model for arrays and some pointer arithmetic. Different from this work, Cheng leveraged potentially unsafe type filtering to accelerate the analysis process and lacks a model for pointer arithmetic.

A unique approach to field sensitivity was taken by Ghiya et al. [30] which, instead of modeling a single field and separate structure objects (the traditional view of field insensitivity), chooses to lump all structure objects together and maintain separate fields. This trades one form of imprecision for another but does not address the complications of maintaining full field sensitivity.

Chapter 5 of this dissertation covers many of the complications in obtaining a field sensitive result. It provides a detailed discussion of safety, provides a baseline model for an offset based field sensitive pointer analysis, and presents algorithms for achieving field sensitivity within a context sensitive analysis.

2.8 Flow Insensitive Analysis

To focus the undertaking enough to allow for extensive development and experimentation within other pointer analysis facets and because true flow sensitivity can substantially increase analysis time, design decisions and algorithms for this dissertation focus on a flow insensitive analysis process. However, Fulcra needs little adaptation to leverage SSA-based partial flow-sensitivity. The following introduces a rudimentary form of the framework used to perform the pointer analysis.

The analysis framework is based on a constraint graph whose nodes represent objects and edges represent assignments. For the sake of this work, the term *constraint* will be used to refer to an assignment as represented by the constraint graph.

Copy	`u := v`
Address	`u := &v`
Load	`u := *v`
Store	`*u := v`

Figure 2.12 Four core constraints for a field- and flow-insensitive analysis.

$$\frac{u := v \quad v := \&w}{u := \&w}$$

$$\frac{u := \&v \quad *\,u := w}{v := w}$$

$$\frac{u := *v \quad v := \&w}{u := w}$$

Figure 2.13 Three derivation rules for a field- and flow-insensitive analysis.

At its simplest there are four assignment constraints, shown in Figure 2.12. Assignments in C programs can be easily broken down into these four constraints. The only caveat is that some assignments require the addition of temporary variables and multiple constraints. For example, the expression `u := **v` must be broken into the following two constraints `u := *tmp` and `tmp := *v`.

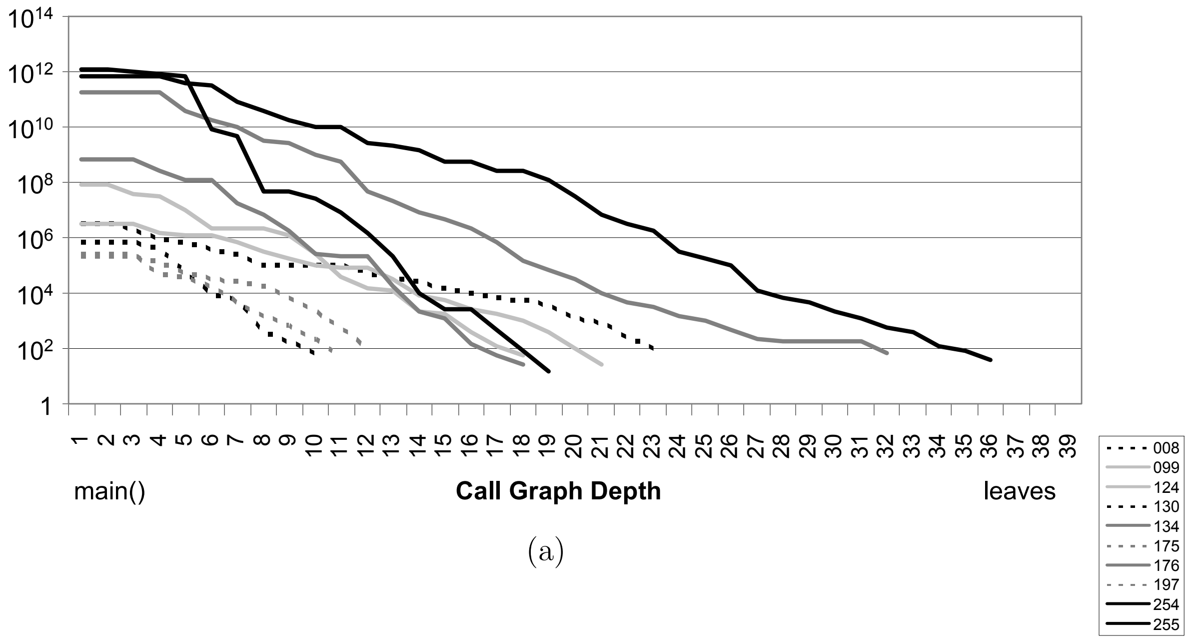
Provided a constraint graph, the three derivation rules in Figure 2.13 can be used to obtain a closure on the constraint graph. This closure is the result of the context-insensitive, flow-insensitive, field-insensitive analysis. Constraints created directly from program expressions will be called *explicit*, while those derived from the analysis process will be called either *derived* or *implicit*.

Note that both copy and address constraints may be derived. For the bulk of this dissertation, the important output of the pointer analysis process is the set of all address constraints because the points-to set of an object is simply the reverse direction of the address constraints (e.g., the assignment of the address of `v` into `u`, `u := &v`, is equivalent to `u` points to `v`).

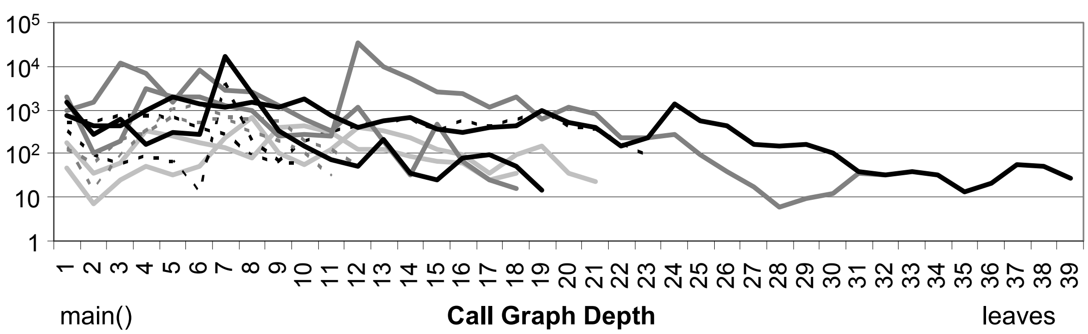
3. THEORETICAL CONTEXT SENSITIVE MODEL

Regardless of the mechanism through which context sensitivity is actually achieved, the end goal is to obtain pointer analysis results equivalent to those obtained if a program's call graph was explicitly expanded before analysis. However, this expansion quickly becomes intractable for even small programs. For example, call graph expansion for small programs from SPEC2000 results in millions of variables while larger programs expand into trillions. Specific data for this is shown in Figure 3.1(a). However, using techniques motivated by the model in this chapter and detailed in Chapter 4 the benefits of full expansion can be obtained by only communicating a compact form of procedural side-effects to callers, and the resultant growth is shown in Figure 3.1(b). The cost of context-sensitivity no longer grows exponentially with program depth but grows and recedes with the characteristics of the procedures. If necessary, refer back to Section 2.4 for a discussion of the basics of context sensitivity.

This chapter presents an approach to obtaining context sensitive results in the form of a two-phase analysis. First Section 3.1 provides an overview of concepts that form the theoretical foundation for the design advocated in this dissertation. Additional coverage of this theoretical model can be found in [22, 23]. Section 3.3 describes the limitations of the theoretical model if implemented and applied to real world programs. Details of the actual analysis model used follow in Chapter 4.



(a)



(b)

Figure 3.1 Effective procedure size when context-sensitivity is achieved using (a) a explicitly expanded call graph and (b) when the effects of callees are represented through the use of compact procedural summaries.

3.1 Model Two-Phase Context Sensitive Algorithm

The high-level rationale behind the two-phase process is best demonstrated by re-examining the code example from Figure 2.5(a). The addresses of `a` and `b` are from two different call sites to `Zulu()` and do not interact in any real execution of the program. However, using a context insensitive analysis, their inputs from `Yank()` and `Xray()` appear to interact at `*p:=q`. The expression `*p:=q` is a side-effect of `Zulu()`. Understanding side-effects is critical to understanding context sensitivity. In particular, removing the sharing of side-effects along different call paths is the central goal of a context sensitive analysis.

As discussed in [22] and proved in [23], no unrealizable data flow can occur if a program lacks procedural side effects. Even though a single procedure may be shared by multiple call paths and, thus, have many different input calling contexts, without side effects interaction among contexts is impossible. For side-effect free programs, there is no difference between a context sensitive and insensitive analysis.

In reality, most programs have procedural side effects. Therefore, to exploit this finding, the two-phase analysis process initially proposed in [22] hoists *specialized* copies of the procedural side effects from callees into callers and, then, deletes the original side effect causing assignments from the callees. The continued hoisting of side effects into procedures higher in the call graph will eventually reach a procedure in which the once side-effect-causing assignments no longer have effects visible outside of their new owner and, thus, are no longer side effects. This upward specialization proceeds until the program is free of all side effects. With this conversion complete, a context-insensitive style analysis can compute a result equivalent to a context sensitive analysis of the original, side-effect containing program. To recap:

1. The bottom-up phase transforms a program into a form that lacks procedural side effects, yet preserves the same overall pointer behavior. The transformation takes place by creating summaries that exactly model the side effects of the original procedure and then inlining specialized copies of the summaries into each call site of that procedure.

2. The top-down phase consists of a single run of the context-insensitive analysis. Since, after the bottom-up phase, the program is free of side effects, the context-insensitive analysis does not experience any unrealizable interprocedural data flow.

3.2 Transforming a Program into a Side-Effect Free Form

Side-effect hoisting takes place through the creation and inlining of procedural summaries. The purpose of a summary is to compactly model all of the side-effects of the original procedure including any visible side effects from its transitive callees. From the perspective of pointer analysis, every call to `foo()` can be replaced by a specialized copy of summary(`foo()`).

Summary construction is performed through a reverse-topological traversal of the acyclic-rendered call graph. Summarization of leaf procedures is performed first. With their summaries constructed, the original side-effects are deleted. Next, specialized copies of the summaries are inlined into their callers, effectively transforming some of those callers into leaf routines themselves. This process continues until all call sites have been replaced by explicit summaries and all side-effects deleted.

Summary size is the single most important factor governing the scalability of the bottom-up phase. Without explicit intervention, summary sizes can grow explosively as the side-effects of callees are incorporated into callers. The following techniques aid in compacting the summary necessary to model the original side-effects.

3.2.1 Criticality

All potential side-effect derivations are identified based on a concept called *criticality*. Informally speaking, an assignment is *critical* if it has the potential to cause a side effect. Two properties, *holding* and *opacity*, are prerequisites to locating critical assignments.

A variable is holding if it may contain a reference to a memory location external to the procedure. For example, a procedure's formal parameters are holding as well as any variable to which a parameter is assigned. In essence, the holding property simulates the effect of a generic, worst-case calling context and helps mark all assignments that may interact with objects external to the procedure. With the knowledge of which variables hold external references, assignments of the form `*u := v`, where both `u` and `v` hold, are critical.

```
whiskey(w,x,y,z) {
    r := z;
    s := &a;
   *x := s;
    t := *y;
   *t := r;
   *a := w;
}
```

(a)

```
summary-whiskey(w,x,y,z) {
    s := &a;
   *x := s;
    t := *y;
   *t := z;
   *a := w;
}
```

(b)

Figure 3.2 Code example (a) for which a context independent summary cannot be formed in terms of only function inputs and (b) a possible summary retaining a reference to `a`.

However, Section 2.4 noted that it is not always possible to form a single, context independent summary in terms of only the procedure parameters. The code example from Figure 2.6 is reproduced in Figure 3.2 but modified to fit the constraint model described in Section 2.8. The assignment `*a := w` will cause a side effect only if the `*x` and `*y` alias. Detecting that `*a := w` is critical and the "conditional" inclusion of its side-effect into the summary is solved by the concept of opacity.

A variable is *opaque* if its contribution to actual data flow is uncertain without specific knowledge about the procedure's calling context. All variables declared outside of the procedure being summarized are opaque. Clearly there is no way to know, local to the procedure, how these variables interact with each other. A local variable can, itself, become opaque if its address interacts with an opaque variable. For example in Figure 3.2, the flow of `&a` through `s := &a` and `*x := s` causes `a` to become opaque because `*x` may be an external, and therefore opaque, location. It it not known how `*x` relates to any

$$\frac{u{:=}{*v} \quad holding(v)}{holding(u)}$$

$$\frac{u{:=}v \quad holding(v)}{holding(u)}$$

$$\frac{{*u}{:=}v \quad v{:=}\&w \quad holding(u)}{opaque(w)}$$

$$\frac{u{:=}\&v \quad opaque(u)}{opaque(v)}$$

$$\frac{param(u) \vee opaque(u)}{holding(u)}$$

$$\frac{u{:=}\&v \quad opaque(v)}{holding(u)}$$

(a)

$$\frac{*u := v \quad holding(u)}{critical(*u := v)}$$

$$\frac{u := v \quad opaque(u)}{critical(u := v)}$$

$$\frac{u := *v \quad opaque(u)}{critical(u := *v)}$$

$$\frac{u := \&v \quad opaque(u)}{critical(u := \&v)}$$

(b)

Figure 3.3 Symbolic derivation of side effects: (a) symbolic effects of calling contexts, and (b) determination of criticality.

other expression that references an opaque location, like `*y`, making the contribution of `a` or, any use of `a` such as `*a`, to data flow unknown. Note that all opaque variables must also be regarded as holding.

The complete derivation rules for holding and opacity, as published in [22], are shown in Figure 3.3(a). Using these two concepts, criticality is extended to also include assignments involving opaque variables. Figure 3.3(b) shows the complete derivation rules for criticality.

With `a` opaque, the condition inclusion of the side effect for the summary of `whiskey()` is handled by considering both `*x := s` and `*a := w`, along with `*t := r`, as critical and using them to seed summary creation. In the end, the the summary of `whiskey()` will include all of the original procedure with the exception of compacting away the copy assignment through `r` and is shown in Figure 3.2(b). The side-effects of the summary retain their dependence on calling context and faithfully model the original procedure.

Recall that opaque variables are treated as being external. For this reason, all modifications of an opaque variable are considered critical and cut from the original procedure. The procedure may still retain assignments that only read from the opaque variable. However, after cutting critical assignments and inlining specialized versions of the summaries, modifications to the opaque variable are no longer performed within the original procedure but, instead, only within its callers; yet reads may still occur within the original procedure. For example, the opacity of `a` made `*x := s` critical in Figure 3.2. This means that `*x := s` will be deleted from the original procedure. Yet it is through `*x := s` that `a` obtains its value leaving `a` and derivations that read from `a` incomplete.

Since opaque variables are treated as external, this gap is filled by making an opaque variable an implicit extra parameter to the procedure. In addition to deleting the critical assignments from `whiskey()`, its call is now modeled as `whiskey(w,x,y,z,a)`. All call sites for `whiskey()` are extended to be `whiskey(-,-,-,-,`$\mathtt{a}_n$`)`, where $\mathtt{a}_n$ is the specialized version of `a` from the summary. The original `a` will now obtain all values from its specialized versions, as it should.

$$\frac{\mathit{traced}(*u := v) \quad u := w \quad \mathit{holding}(w)}{\mathit{traced}(*w := v)}$$

$$\frac{\mathit{traced}(*u := v) \quad u := *w \quad \mathit{holding}(w)}{\mathit{add}(*u := v) \wedge \mathit{traced}(u := *w)}$$

(a) stuck left-hand

$$\frac{\mathit{traced}(*u := v) \quad v := w \quad \mathit{holding}(w)}{\mathit{traced}(*u := w)}$$

$$\frac{\mathit{traced}(*u := v) \quad v := \&w}{\mathit{add}(*u := v) \wedge \mathit{traced}(v := \&w)}$$

$$\frac{\mathit{traced}(*u := v) \quad v := *w \quad \mathit{holding}(w)}{\mathit{add}(*u := v) \wedge \mathit{traced}(v := *w)}$$

(b) stuck right-hand

$$\frac{\mathit{traced}(*u := v) \quad \mathit{input}(u) \quad \mathit{input}(v)}{\mathit{add}(*u := v)}$$

(c) stuck both sides

$$\frac{\mathit{traced}(u := v) \quad v := \&w}{\mathit{traced}(u := \&w)}$$

$$\frac{\mathit{traced}(u := v) \quad v := *w \quad \mathit{holding}(w)}{\mathit{traced}(u := *w)}$$

$$\frac{\mathit{traced}(u := v) \quad v := w \quad \mathit{holding}(w)}{\mathit{traced}(u := w)}$$

$$\frac{\mathit{traced}(u := v) \quad \mathit{input}(v)}{\mathit{add}(u := v)}$$

$$\frac{\mathit{traced}(u := \&v)}{\mathit{add}(u := \&v)}$$

(d)

$$\frac{\mathit{traced}(u{:=}*v) \quad v{:=}w \quad \mathit{holding}(w)}{\mathit{traced}(u{:=}*w)}$$

$$\frac{\mathit{traced}(u := *v) \quad v := *w \quad \mathit{holding}(w)}{\mathit{add}(u := *v) \wedge \mathit{traced}(v := *w)}$$

$$\frac{\mathit{traced}(u := *v) \quad \mathit{input}(v)}{\mathit{add}(u := *v)}$$

(e)

Figure 3.4 Backtracing for (a), (b), (c) store, (d) plain and address, and (e) load assignments.

3.2.2 Backtracing

Once located, critical assignments seed a process called *backtracing* which creates the actual summaries by compactly completing any necessary data flow from the procedure's inputs to the critical assignments.

The bulk of the backtracing process is straightforward. Referring back to the example, the critical assignment `*x := s` and its input `s := &a` cannot be more concisely represented, so both are added to the summary. However, `*t := r` and `r := z` yield `*t := z`. This models the original data flow while reducing overall summary size. Since their total contribution to data flow is known, all nonopaque variables, with the exception of parameters and temporaries required by the constraint model, can be compacted away. However, because it is unclear how opaque variables contribute to the final data flow, *all* potential modifications of opaque variables must be reflected in the summary. The complete backtracing derivation rules are shown in Figure 3.4.

With the summaries formed, the critical assignments are deleted from the original procedure, leaving it free of side effects while maintaining the effect of the original assignments.

3.3 Limitations of Theoretical Model

The model described has both an elegance in its approach to context sensitivity, reducing it to the task of transforming a program into an equivalent yet side-effect free form, as well as practical shortcomings that obstruct its direct use in an pointer analysis framework.

One unfortunate characteristic of this approach to context sensitivity is that it is an explicit transformation. While the concept of converting a program into a side-effect free form is useful for proofs it is also permanent and inflexible. This becomes particularly apparent when one considers that, in the presence of indirect function calls, the analysis process is iterative and must restart from each new call graph. The explicitness of the transformation imposes a strict order between the bottom-up and top-down phases requiring each iteration to delete of all derivations, reinsert the deleted side-effects, and perform a renewed bottom-up process. Effectively, this reruns the analysis process every time the top-down phase discovers a call graph update.

The approach presented in this dissertation breaks this dependence between the bottom-up and top-down processes. It transforms the explicit deletion of critical assignments into an implicit one and permits the complete retention of all previously derived pointer relationships. With the capability to run the bottom-up and top-down phases in any order, even simultaneously, a more efficient call graph discovery process can be used that requires fewer iterations, sometimes none, in the presence of indirect function calls.

As mentioned earlier, program expansion due to the explicit inclusion of summaries and therefore summary size is the primary contributor to the cost of the bottom-up process, if not to the entire analysis process. However, the works [22, 23] make little mention of the effects of global variables and large recursive call graph cycles. Global variables are always opaque because they exist outside the scope of a procedure; their final contributions to side-effects are unknown until the analysis is complete. Handled naïvely, global variables build up within the summaries and the resultant dead weight can bring the bottom-up process to a halt. Furthermore, the cost of summarization is a function of the procedure size, and recursive cycles, as described in Section 2.4, are treated as one large metaprocedure. If particularly large, the cost of their summarization can dominate the analysis time.

The same approach that simplifies the iterative analysis process also enables a seamless model for selective context sensitivity. Selective context sensitivity means that some variables can be treated in a context-sensitive manner while others in a context-insensitive one. In addition to being an interesting and little investigated concept, it can be used to completely remove the overhead of global variables from the summarization process without loss of analysis accuracy. Furthermore, it provides a flexible mechanism in which to handle the summarization of program recursion and to control heap specialization, both critical to a real pointer analysis implementation.

Another topic not covered by the original model is the coexistence of field sensitivity and context sensitivity. While seemingly orthogonal and bringing its own trade-offs and complexities, field sensitivity complicates the construction of context-sensitive pointer analysis algorithms and makes the process far more resistant to efficient computation. The rules governing criticality detection, summarization, and redundancy removal become far more involved. Furthermore, procedural summarization becomes sensitive to an explosion in size. This dissertation covers the details necessary to allow the peaceful coexistence of both field sensitivity and context sensitivity.

Finally, because the ideas in this dissertation exist within an implemented pointer analysis framework, many of the discussions and algorithms try to communicate enough details to facilitate easy use by other designers of pointer analysis systems. While Fulcra is, to this author's knowledge, the most efficient pointer analysis system of its kind to date, its development has been incrementally driven by experimentation and discovery.

Thus, there is room for the subsequent development of a more flexible and efficient framework. This dissertation will hopefully provide a blueprint to guide the component design many other frameworks.

4. CONTEXT SENSITIVITY

Chapter 3 walked through an approach to context-sensitivity which provides a satisfying conceptual model but has many shortcomings when one desires to transform the model into an efficient implementation. This chapter starts by revisiting the concept of interprocedural data flow and its relationship to context-sensitivity. It then proposes an analysis framework based on qualified constraints, i.e., constraint attributes that influence the derivation process, and which forms a comprehensive, flexible, and efficient backbone to the context-sensitive analysis process. Finally, it presents detailed algorithms for the analysis framework and experimental evaluation of the approach.

4.1 Interprocedural Data Flow Revisited

With respect to context sensitivity, two major concepts have been discussed so far:

1. Regardless of the mechanism through which context sensitivity is actually achieved, the end goal is to obtain pointer analysis results equivalent to those obtained if a program's call graph were fully expanded, analyzed context insensitively, and then the results for each instance of a procedure annotated back to the original.

2. No unrealizable interprocedural data flow can occur if a program lacks procedural side effects. Even though a single procedure may be shared by multiple call

paths and, thus, have many different input calling contexts, without side effects the different contexts cannot interact.

Chapter 3 walked through how [22, 23] leveraged these ideas by creating an explicit transformation from a program with side effects into an equivalent, side-effect free one. While elegant, it turns out that such a brute force explicit transformation is both problematic and, in fact, unnecessary.

Figure 4.1 will be used to walk through examples of interprocedural data flow as it pertains to context sensitivity. Figure 4.1(a), (b), and (c) all depict the same call graph in two forms. The solid lines represent calls. The first (left) is a typical depiction of a call graph where `A()` calls `B()` and `C()` which each, in turn, call `D()`. The second (right) is a fully expanded call graph where procedure `D()` has been broken into its two instances $\mathtt{D}_1$ and $\mathtt{D}_2$`()`. In each call graph, gray circles represent some variable within the respective procedure and dotted lines represent data flow between the variables. The desire of an efficient context-sensitive analysis is to efficiently context-sensitively analyze a program in the form of the left call graph without resorting to analyzing the call graph to the right.

All pointer derivations start with constraints generated directly from the program itself, or explicit constraints. Explicit constraints are combined with each other to yield implicit constraints which may be subsequently combined with other explicit and implicit constraints. At some point, a context-insensitive analysis program must generate an implicit constraint not generated had a fully expanded call graph been analyzed. To

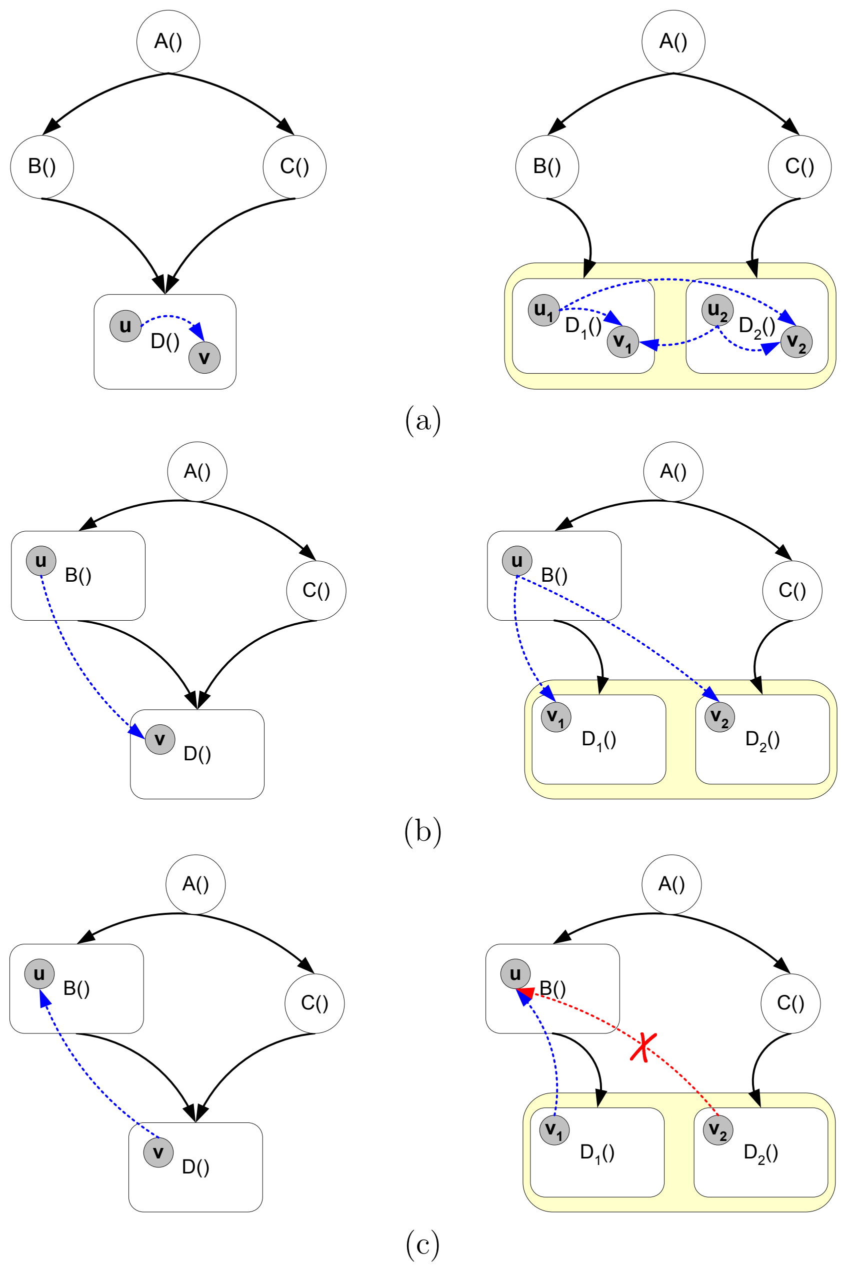


Figure 4.1 Three fundamental forms of interprocedural data flow: (a) local data flow from `u` to `v` termed *horizontal*, (b) interprocedural data flow from caller `B()` to callee `D()` termed *downward*, and (c) interprocedural data flow from callee `D()` to caller `B()` termed *upward*.

locate the point of failure, Figures 4.1(a), (b), and (c) walk through the three forms of interprocedural data flow and map the scenarios to both the original view of the program, left diagram, and the fully expanded version, right diagram.

The first scenario is intraprocedural data flow, and such derivations will be called *horizontal* derivations. Figure 4.1(a) shows a stylized model for this data flow. In the original call graph the constraint flows from `u` to `v`. However, when mapped to the expanded call graph, the effect is data flow from both instances of `u` to both instances of `v`. The dataflow through v_1`:=` u_1 and v_2`:=` u_2 are clearly valid because an intraprocedural constraint derived in one instance should be derived for all instances. While seemingly problematic, v_1`:=` u_2 and v_2`:=` u_1 are, in fact, valid and do not effect the end results. When the points-to results are annotated back to the original procedure `D()`, all specialized instances of each variable are collapsed into a single representative. For example, once the analysis is complete a query about the contents of `v` is a query against the cumulative contents of its instances,v_1 and v_2. Thus, if only local data flow exists, the eventual combination of the two instances of `v` and `u` means that there is no benefit in ever keeping them separate. Thus horizontal data flow can be safely computed on original, nonexpanded procedures.

The second scenario is interprocedural data flow from callers to callees, called *downward* derivations. Figure 4.1(b) shows a stylized model for this data flow. In the original call graph the constraint flows from `u`, in `B()`, to `v`, in `D()`. In the expanded call graph, this maps to data flow from `u`, in `B()`, to both instances of `v`, even though v_2`()` belongs to the `C()`'s instance of `D()`. Data flow due to the constraint $\texttt{v}_1$`:= u` is valid. Similar to the horizontal data flow, $\texttt{v}_2$`:= u` is valid as well because, again, the annotation back to the original procedure `D()` will effectively recombine the instances of `v`. Downward data flow can also be safely computed on the original, nonexpanded procedures.

The third scenario is interprocedural data flow from callees to callers, called *upward* derivations. Figure 4.1(c) shows a stylized model for this data flow. In the original call graph the constraint flows from `v`, in `D()`, to `u`, in `B()`. In the expanded call graph, this maps to data flow from both instances of `v` to `u`. The data flow from $\texttt{v}_1$ to `u` is valid since $\texttt{D}_1$`()` is the instance matching `B()`. However, data flow from $\texttt{v}_2$ to `u` is unrealizable. The variable `u` should not obtain the contents of `v` from `C()`'s call to TTD(). Unlike for the horizontal and downward cases, no instances of `u` exist in `C()` whose inclusion with the instance of `u` in `B()` will validate this data flow. This initial point of failure when analyzing the unexpanded call graph is the derivation of upward constraints.

There are two key insights that can be taken from this observation:

1. The first instance of an upward constraint is always an upward copy constraint. This is easily shown as store and load constraints are never derived and new address constraints cannot be derived without a copy constraint.

2. All upward copy constraints result from critical assignments. [1] Unrealizable data flow is not due to the critical assignments themselves but, instead, due to the use of the resultant upward copy constraints to make further derivations.

These insights mean that two options exist for preventing unrealizable data flow that do not entail the deletion of critical assignments following summarization.

- Option 1 - Don't derive upward copy constraints
- Option 2 - Don't derive upward address constraints (i.e., derive upward copy constraints but don't use them)

For now, assume the use of option 1. If the original program is analyzed but upward constraints are disallowed, the result would be a strict subset of a context-sensitive analysis where all side-effects were ignored. This is clearly not the goal, but is extremely close. The goal of the bottom-up summarization process detailed in Section 3.1 was to hoist side-effects from callees for the creation of specialized, local copies in callers. However, it had to explicitly delete the original side-effect producing assignments to prevent them from corrupting the result.

If the same bottom-up summarization process were used, but instead of deleting side-effect producing assignments upward constraints were disallowed, the same context-sensitive result could be obtained. Furthermore the dependence that existed between the bottom-up and top-down phases is now broken. A global derivation step can be

[1] Upward copy constraints can also result from procedure return statements. This case is very similar to critical assignments and discussion will be left for Section 4.8

performed at any time and the only side-effects taken into account will be those already captured by the included summaries. If a call graph update is discovered forcing the need for another iteration, all implicit constraints can be recycled as long as the summarization process looks at only explicit and implicit horizontal constraints. A benefit that leverages the independence of the two phases is covered in Section 4.4.

4.2 Partial Context Sensitivity

The three forms of interprocedural data flow – horizontal, downward, and upward – are sufficient to describe a flexible approach to context sensitivity. In fact, the derivation of upward data flow appears important only in that it should be avoided. However, with the addition of one other kind of data flow, upward copy constraints become a valuable asset and yield a far more general analysis process.

While equivalent, if option 2 is selected instead of option 1, the following two observations can be made:

1. With the creation of upward copy constraints, the only difference between obtaining a context-sensitive result and an insensitive result is whether or not the upward copy constraints are used for subsequent derivations.

2. If some upward copy constraints were selectively used, a partially context sensitive result would be obtained.

At first, it may seem wasteful to contaminate an otherwise context sensitive result by using upward copy constraints. However, if done methodically, such use is, in fact, invaluable. If (1) a set of upward constraints corresponding to a particular critical assignment were selected for use, (2) all address constraints that result from their use were recorded, and (3) all subsequent derivations made use of every upward constraint they encountered, then the critical assignment would not have to be included in the summary. The consequence of the three actions is the treatment of a particular side-effect context-insensitively.

The distinction between context-sensitive data flow, data flow that can use upward constraints, and context-insensitive data flow, that must be communicated through summaries, is made via a new kind of derivation, called a *global* derivation. By allowing global constraints to freely flow upward, it is unnecessary to include the effects of data flow through context-insensitive variables into summaries. While this may lead to unrealizable derivations it can also reduce the overhead of the cost summarization process.

Figure 4.2 shows the effect of global constraints and partial context-sensitivity using a stylized depiction of the interprocedural data flow. The call graph in Figure 4.2 shows only the expanded version of the call graph. The variable `g` in `F()` is being treated context insensitively. In doing so, `g` effectively spans all instances of `F()`. For the example, it is assumed that the contents of `g` interact with a side-effect visible to calling procedures `E()`, `C()`, and `A()`. The contents of `g` flow freely interprocedurally to both `z` and `y` and, subsequently, to `x` the need for a summary. The data flow continues, downward now, to both instances of `v` in `D()`. As previously discussed, downward flow is always valid. However, note that data flow from $\mathtt{v}_2$ into `u` should not occur (just as in Figure 4.1(c)).

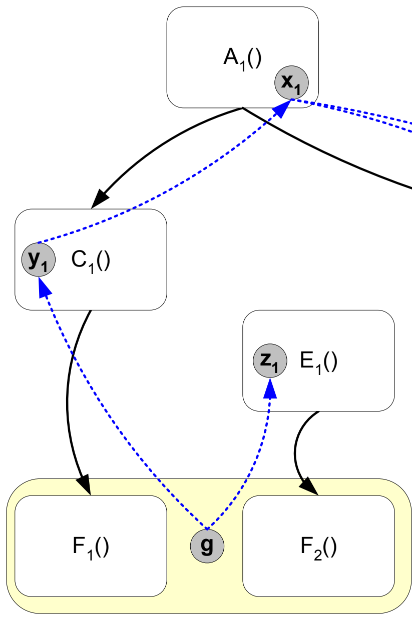


Figure 4.2 Example data flow caused by the use of a globally qualified constraint.

While the global constraints due to `g` can flow freely upward any downward derivations must lose this freedom. From another point of view, loss of context-sensitivity in `F()` should only cause the unrealizable interaction of data flowing from its callers `E()`, `C()`, or `A()`. It should not cause unrealizable mixing between `A()` and `B()` through `D()`. Benefits of leveraging partial context sensitivity are detailed in Section 4.5.

4.3 Qualified Constraints

Qualifiers, essentially annotations, on constraints are used to guide the derivation process. As outlined in Sections 4.1 and 4.2, there are four qualifiers: Horizontal (HS) qualifiers communicate purely intraprocedural data flow, Downward (DN) communicate inter-procedural data flow from callers to callees, Upward (UP) communicate inter-procedural data flow from callees to callers, and Global (GBL) communicate inter-procedural data

Initial Qualifiers

$$\frac{\texttt{v:=\&w} \quad \mathrm{ci}(w)}{GBL}$$

$$\frac{\text{formal-actual-param}(\texttt{v:=w})}{DN}$$

$$\frac{\text{formal-actual-return}(\texttt{v:=w})}{UP}$$

$$\frac{\text{all others}}{HZ}$$

Q-Matrix			`u := &w`		
kind(u)			**HZ**	**DN**	**GBL**
cs	`*u:=v`	**HZ**	HZ	UP	GBL
cs	`v:=*u`	**HZ**	HZ	DN	GBL
cs	`v := u`	**HZ**	HZ	DN	GBL
cs	`v := u`	**DN**	DN	DN	DN
cs	`v := u`	**UP**	**spurious**		GBL
cs	`v := u`	**GBL**	GBL	GBL	GBL
ci	`*u:=v`	**HZ**	GBL	GBL	GBL
ci	`v:=*u`	**HZ**	GBL	GBL	GBL
ci	`v:=u`	**HZ**	GBL	GBL	GBL
ci	`v:=u`	**DN**	DN	DN	DN
ci	`v:=u`	**UP**	GBL	GBL	GBL
ci	`v:=u`	**GBL**	GBL	GBL	GBL

Figure 4.3 Initial conditions and derivation rules for qualifier-based context sensitivity.

flow context-insensitively. Horizontal and Downward constraints are sufficient if one only wants a simple, though potentially expensive context-sensitive analysis framework. The addition of Upward and Global qualifiers permits a more flexible, partially context-sensitive analysis process. This section presents the details for using a qualified constraint model to compute a flexibly context-sensitive analysis result.

The derivation rule governing constraint qualifiers can be concisely represented (and even implemented) using the qualifier matrix shown in Figure 4.3. The first part of the figure shows the rules for assigning initial qualifiers to constraints. Almost all initial qualifiers are horizontal. There are three exceptions. First, copy assignments representing mappings from actual parameters of a caller to the formal parameters of a callee procedure are downward. The analogous copy assignment from the return statement of a callee to the assigned return variable of the caller obtains an upward qualifier. Finally, if a technique described in Section 4.5 to reduce the summarization overhead due to global variables is used, address constraints for global variables obtain a global qualifier.

The derivation matrix shows the new qualifier obtained given a particular address constraint, copy/load/store constraint, and common variable. The *kind*(`u`) reflects whether `u` is being treated in a context-sensitive or -insensitive way. The marking "spurious" means that the derivation is not allowed to proceed. For example, if a downward address constraint `u:&w` is combined with a downward copy constraint `v:=u`, an address constraint `v:=&w` is derived with a downward qualifier regardless of `u` being treated in a context-sensitive or -insensitive way.

To make better sense of the derivation rules, the matrix can be broken down into a few smaller parts. The upper half is for a context-sensitive analysis and the lower for a context-insensitive analysis. Since load and store constraints are never derived, they are always horizontal. The following covers the key points:

- Horizontal constraints combined with each other result in more horizontal constraints because intraprocedural constraints cannot, by themselves, derive interprocedure ones.

- A derivation involving a downward constraint, with one exception, always derives another downward constraint. If an assignment flows from a caller to a callee, the result of a combination with either a horizontal or another downward will still result in data flow from a caller to some subsequent callee. The exception is when a downward address constraint `u:=&w` combines with a store constraint `*u:=v`. By dereferencing `u` the data flow reverses direction resulting in the only derivation of an Upward constraint.

- Upward constraints can only be used by Global (context-insensitive) constraints.

- Use of Global constraints in a derivation almost always result in other global constraints. Once a derivation is context-insensitive, it can flow freely upward to callers, horizontally with procedures, as well as combine with other global constraints. However, "context" is regained whenever data flow goes from a caller to a callee. This is always the case, global or not. The use of a downward copy constraint always results in another downward constraint, even for global address constraints.

4.3.1 Qualified constraint example

The following example will be used to walk through the derivation process for the simple program first introduced in Figure 2.5. A version of this program modified to fit the constraint model is shown in Figure 4.4(a) and the analysis process is shown in Figure 4.4(b). For this example all variables will be treated in a context-sensitive manner. The initial constraints consist of all constraints created explicitly from the program text. Constraints c_1 through c_5 are purely intraprocedural and thus horizontal. Constraints c_6 through c_9 model actual to formal parameter passing and are downward.

This bottom-up summarization phase will replace the calls to `Zulu()` with its side-effects. `Zulu()`'s side-effect is `*p := q` which maps to constraints c_{10} and c_{11} when

```
    Yank() {
1:      z := &a;
2:      y := &1;
3:      Zulu(z,y)
    }

    Xray() {
4:      x := &b;
5:      w := &2;
6:      Zulu(x,w)
    }

    Zulu(p,q)
7:      *p := q
    }
```

(a)

Initial Constraints
HZ (c_1 `z := &a`, c_2 `y := &1`,
c_3 `x := &b`, c_4 `w := &2`, c_5 `*p := q`)
DN (c_6 `p := z`, c_7 `p := x`, c_8 `q := y`, c_9 `q := w`)

Bottom-Up Summarization Constraints
Summary of `Zulu()` found to be `*p := q`
Call at line 3 replaced with HZ (c_{10} `*z := y`)
Call at line 6 replaced with HZ (c_{11} `*x := w`)

Derived Constraints
From `Yank()`
$c_1 + c_{10} \rightarrow$ HZ (c_{12} `a := y`)
$c_{12} + c_2 \rightarrow$ HZ (c_{13} `a := &1`)
$c_1 + c_6 \rightarrow$ DN (c_{14} `p := &a`)
$c_2 + c_8 \rightarrow$ DN (c_{15} `q := &1`)

From `Xray()`
$c_3 + c_{11} \rightarrow$ HZ (c_{16} `b := w`)
$c_{16} + c_4 \rightarrow$ HZ (c_{17} `b := &2`)
$c_3 + c_7 \rightarrow$ DN (c_{18} `p := &b`)
$c_4 + c_9 \rightarrow$ DN (c_{19} `q := &2`)

From `Zulu()`
$c_5 + c_{14} \rightarrow$ UP (c_{20} `a := q`)
$c_5 + c_{18} \rightarrow$ UP (c_{21} `b := q`)
$c_{20} + c_{15} \rightarrow$ spurious
$c_{20} + c_{19} \rightarrow$ spurious
$c_{21} + c_{15} \rightarrow$ spurious
$c_{21} + c_{19} \rightarrow$ spurious

(b)

Figure 4.4 Illustration of the use of qualifiers in a context sensitive analysis process. (a) Example program and (b) the initial conditions and constraint derivation process.

substituted for the actual parameters. The derivations made during the intraprocedural and top-down derivation processes for `Yank()` and `Xray()` are shown explicitly and derived the desired relations `a := &1` and `b := &2`. The side-effect of `Zulu()` causes the derivation of two upward copy assignments. However, the side-effect is effectively disabled since all variables are being treated context-sensitively disallowing the combination of downward and upward constraints.

4.3.2 Qualified constraint mix

Table 4.1 shows the mix of qualifiers seen for context-sensitive analyses of the application suite. The percentage of upward constraints is always very small. This is the result of two factors. First, only copy constraints can be upward and address constraints usually far outweigh copy constraints. Second, to an extent, upward constraints reflect the amount of data flow due to nonglobal side-effects in an application. It makes sense that there would be a large quantity of intraprocedural, caller-to-callee, and global data flow further reducing the percentage of upward constraints measured.

4.4 Modified Two-Phase Algorithm

Section 4.1 introduced the idea that, by using qualified constraints, dependence that existed between the bottom-up and top-down phases disappears. This means that a low-cost partial solution can be obtained at any time by performing a global, top-down derivation. The result is partial in the sense that some (if not all) procedural summaries

Table 4.1 Mixture of constraint qualifiers observed.

Benchmark	% HZ	% DN	% UP	% GBL
008.espresso	5	64	2	29
023.eqntott	42	40	4	15
099.go	41	54	0.1	5
124.m88ksim	34	52	1	13
129.compress	46	37	0	17
130.li	10	6	0.3	84
132.ijpeg	40	54	5	2
134.perl	2	16	3	80
164.gzip	38	51	1	9
175.vpr	29	43	4	25
176.gcc	2	22	0.8	75
181.mcf	43	37	5	15
186.crafty	31	56	1	13
197.parser	18	33	2	48
253.perlbmk	1	4	0.1	95
254.gap	4	11	0.4	85
255.vortex	4	48	0.6	48
256.bzip2	44	50	0.6	47
300.twolf	25	33	2	40
gsmdec	42	46	1	10
gsmenc	42	46	1	10
jpegdec	38	55	6	1
jpegenc	39	55	4	2
h263dec	34	56	1	8
h263enc	38	58	3	1
mpeg2dec	35	54	0.8	11
mpeg2enc	31	63	0.5	5
mpg123	40	47	1	12
jpeg2kdec	23	70	7	0.5
mpeg4dec	14	81	5	0.4

may not yet exist. It is low cost in the sense that all derivations made would have had to be made at some point and they do not interfere or complicate subsequent steps in the analysis process. The process of obtaining an early, partial solution will be called a *solution preview*, though note that, while it is a preview, the results are not discarded.

One overhead of a summarization-based context-sensitive analysis is the cost of resummarization of procedure along call paths to locations where new indirect function call targets are discovered. If an oracle provides an accurate call graph, a procedure would never have to be resummarized. Any process that accelerates the discovery of call targets, in particular before the bottom-up summarization phase, can reduce the cost of the analysis process. While an oracle does not usually exist, any, even partial, information that can help resolve indirect function call targets is beneficial. A solution preview does exactly this.

4.4.1 Solution preview

The typical two-phase analysis process is modified to first iteratively perform a solution preview (i.e., make all possible derivations without performing any summarization). Many indirect function call targets are passed downward from caller to callee so, often, some targets can be resolved without the complete model of procedural side-effects. If new targets are discovered, the call graph is updated and the solution preview is extended to include the new procedures. The process continues until no more targets are discovered.

Next, a typical bottom-up summarization step is performed. Following this, the solution preview process repeats. Once a summarization step has been performed, the analysis process is complete if the solution preview does not discover new targets (i.e., the solution preview is in fact the solution). In many cases, only one bottom-up sweep is necessary even though an application contains indirect function calls. Table 4.2 lists the number of iterations using a typical two-phase analysis algorithm and the number necessary when call graph discovery is accelerated using solution previews. Data is omitted for programs lacking indirect function calls.

4.4.2 Constraint recycling

Qualified constraints also allow the complete retention of derivations from one analysis iteration to the next. Downward derivations pertain only to the top-down process. Global and upward derivations are for context-insensitive derivations. Furthermore, constraints are never deleted, as in the model described in Chapter 3. Therefore, all derivations can be retained as long as the bottom-up summarization phase looks at only horizontal derivations. By doing so, the number of derivations tends to quickly taper off from one iteration to the next. Table 4.2 shows the resultant speedup obtained by an implementation that recycles its constraints between iterations against one that must rederived everything each iteration. Note that the cost can be more than double because constraints must both be deleted and rederived.

Table 4.2 Comparison of the number of iterations necessary for a traditional two-phase algorithm and one leveraging a solution preview, along with the analysis speedup when constraints are recycled.

Benchmark	Orig Iter	New Iter	Recycle Speedup
008.espresso	2	1	-
023.eqntott	2	1	-
099.go	-	-	-
124.m88ksim	4	3	2.4
129.compress	-	-	-
130.li	4	3	2.2
132.ijpeg	7	4	3.2
134.perl	2	1	-
164.gzip	2	1	-
175.vpr	2	1	-
176.gcc	4	3	1.7
181.mcf	-	-	-
186.crafty	-	-	-
197.parser	2	1	-
253.perlbmk	4	3	3
254.gap	4	2	2.1
255.vortex	3	2	2.8
256.bzip2	-	-	-
300.twolf	-	-	-
gsmdec	2	1	-
gsmenc	2	1	-
jpegdec	7	4	2
jpegenc	6	4	2.7
h263dec	-	-	-
h263enc	-	-	-
mpeg2dec	-	-	-
mpeg2enc	-	-	-
mpg123	3	2	2.2
jpeg2kdec	4	2	1.1
mpeg4dec	-	-	-

```
    Quebec() {
1:     a := &oscar;
2:     b := &nov;
3:     Papa(a,b)
    }

    Papa(c,d) {
4:     (*c)(d);
    }

    Oscar(e) {
5:     (*e)();
    }

    Nov() { }
```

Figure 4.5 Sample program used to demonstrate the benefits of a modified two-phase algorithm.

4.4.3 Example

The code in Figure 4.5 will be used to illustrate this process and its benefit. For both clarity and brevity, the procedures used for the example show only assignments related to call graph generation.

The program consists of a direct call from `quebec()` to `papa()` which calls `oscar()` indirectly, which then indirectly calls `nov()`. If analyzed in a typical two-phase approach, an initial call graph would be built consisting of `quebec()` and `papa()`. The first bottom-up process would ensue and, following it, a top-down phase would derive `c := &oscar` and that the indirect call at line 4 calls `oscar()`. The call graph would be updated and a second bottom-up and top-down phase would be performed. Now, `e := &nov` would be derived and the call graph extended to include the indirect call from line 5 in `oscar()` to `nov()`. A third and final bottom-up and top-down phase would be performed. In the end, three expensive bottom-up analysis steps must be performed.

Instead, the modified two-phase process requires only a single bottom-up phase. As before, an initial call graph would be built consisting of `quebec()` and `papa()`. The solution preview from the top-down derivations would show that `c := &oscar` and that the indirect call at line 4 calls `oscar()`. A second solution preview starting at `oscar()` would show that `e := &nov` and would discover the the indirect call from line 5 in `oscar()` to `nov()`. A third solution preview at `nov()` would not reveal any new calls. The first and only bottom-up process would be performed ignoring any of the downward derivations, such as `c := &oscar`, and would be followed by a final top-down phase.

4.5 Applying Partial Context Sensitivity

Section 4.2 introduced the idea of controlling the amount of context sensitivity by treating side-effects involving some variables in a context-sensitive manner while the side-effects others in a context insensitive way. This selection is made before the summarization of a procedure containing data flow explicitly involving the variable. A variable can go from context-sensitive to context-insensitive but not the reverse because context-insensitive derivations will have already been made.

Side-effects that directly or indirectly involve a context-insensitive variable get communicated through the context-insensitive derivation of global constraints and, thus,

```
     Yank() {
1:      z := &a;
2:      y := &1;
3:      Zulu(z,y)
     }

     Xray() {
4:      x := &b;
5:      w := &2;
6:      Zulu(x,w)
     }

     Zulu(p,q)
7:      *p := q
     }
```

(a)

Initial Constraints

HZ (c_1 `z := &a`, c_2 `y := &1`,
c_3 `x := &b`, c_4 `w := &2`, c_5 `*p := q`)

DN (c_6 `p := z`, c_7 `p := x`, c_8 `q := y`, c_9 `q := w`)

Bottom-Up Summarization Constraints

Summary of `Zulu()` found to be nothing

Calls at line 3 and 6 effectively deleted

Derived Constraints

From `Yank()`

$c_1 + c_6 \rightarrow$ DN (c_{14} `p := &a`)

$c_2 + c_8 \rightarrow$ DN (c_{15} `q := &1`)

From `Xray()`

$c_3 + c_7 \rightarrow$ DN (c_{18} `p := &b`)

$c_4 + c_9 \rightarrow$ DN (c_{19} `q := &2`)

From `Zulu()`

$c_5 + c_{14} \rightarrow$ UP (c_{20} `a := q`)

$c_5 + c_{18} \rightarrow$ UP (c_{21} `b := q`)

$c_{20} + c_{15} \rightarrow$ GBL (c_{22} `a := &1`)

$c_{20} + c_{19} \rightarrow$ GBL (c_{23} `a := &2`)

$c_{21} + c_{15} \rightarrow$ GBL (c_{24} `b := &1`)

$c_{21} + c_{19} \rightarrow$ GBL (c_{25} `b := &2`)

(b)

Figure 4.6 Illustration of the derivation process when the variable `q` is treated context insensitively in (a) an example program and (b) the initial conditions and constraint derivation process.

these side-effects do not have to be included into summaries. Ignoring the possibly negative accuracy impact of treating some parts of a program context-insensitively, this provides a way to selectively reduce summarization overhead.

The same program from Figure 4.4(a) will be used to demonstrate the effect of marking a variable for context-insensitive treatment. Figure 4.6(a) reprints the example and Figure 4.6(b) walks through the new analysis process where `q` is treated context-insensitively. Since `q` is the only source to the side-effect in `Zulu()` the bottom-up summarization phase yields an empty summary and the calls to `Zulu()` are effectively deleted.

During the top-down phase, `q` and `p` obtain both `&1` and `&2` via downward constraints and the upward constraints c_{20} and c_{21} are derived just as before in Figure 4.4(b). However, since `q` is CI, the c_{20} and c_{21} derive global constraints resulting in c_{22} through c_{25}, which complete a context-insensitive result.

While Figure 4.6 showed that summary size can be reduced by marking some variables as CI, it also showed that this may have a negative impact on the accuracy of the result. However, side-effects directly or indirectly involving global variables cannot benefit from context sensitivity, making their inclusion into summaries both costly and superfluous. The effects of nonglobal variables are specialized for a particular call site. Specialization essentially renames the variable to isolate its data flow. However, a global variable has global scope and it cannot be renamed, meaning that data flow involving some global variable `G` in a summary will still involve `G` for every instance of that summary. Global variables are context-insensitive by their very nature.

Fulcra's ability to leverage selective context sensitivity allows it to mark all global variables as CI, thereby removing them from the cost of summarization yet providing an identical result. The isolation of global variables from summarization is called *Global Variable Isolation* or *GISO*. Table 4.3 shows a select set of analysis times without and with GISO and shows that GISO significantly decreases the cost of performing a context-sensitive analysis.

Table 4.3 Analysis time necessary when global variables are included into summaries.

Benchmark	Without	GISO (noheap)
008.espresso	INF	6
130.li	INF	4
132.ijpeg	70	6
134.perl	INF	20
176.gcc	INF	hrs
253.perlbmk	INF	hrs
254.gap	INF	800
255.vortex	200	38
300.twolf	5	1

Even with GISO, the cost of performing a context sensitive analysis is still quite high for the applications `176.gcc`, `253.perlbmk`, and `254.gap`. This cost is due to the overhead of summarizing very large metaprocedures due to immense recursive cycles involving hundreds of procedures.

Recall from the background in Section 2.4 that recursion is handled by merging all procedures involved in a recursive cycle into a single, large metaprocedure. The cost of the summarization process has a worst-case cubic complexity with procedure size but, due to the nature of most procedures, is generally linear. Given the common-case of linear, if 100 procedures are merged together a procedure 100 time the size must be summarized once for each merged procedure, a 100-fold increase in cost. This means that the minimum impact of recursion involving a n procedures is roughly a factor of n and a worst-case of increase of a factor of n_4.

Selective context sensitivity can be used to mark procedures in the recursive cycles as context insensitive, thereby eliminating their summarization overhead. Data specific to this will not be shown here since the comprehensive data in Section 6 includes this feature. However, it reduces the cost of analysis of TT176.gcc and `253.perlbmk` from hours to seconds.

4.6 Summaries Containing Opaque Variables

Chapter 3.1 described how, if opaque variables are encountered during the summarization process, procedure parameters are extended to include the opaque variables as additional parameters. This permits opaque variables to obtain their appropriate values while enabling the deletion of all critical assignments involving opaque variables.

While the use of qualifiers removes the necessity to delete constraints, the summarization of opaque variables presents an analogous problem. Refer back to procedure `whiskey()` from Figure 3.2. When combined with a downward address constraint from some caller, for example `x := &m`, the assignment `*x := s` results in an upward copy constraint `m := s`. Since `m := s` is upward it cannot be combined with the horizontal address constraint `s := &a` preventing `a` from obtaining its value.

The derivation blocking due to upward constraints creates the same problem that the deletion of critical assignments caused for the theoretical model. Similar to adding opaque variables to a procedure's parameters, a downward copy constraint is added from all specialized copies of opaque variables (created by summary instantiation) to their original instance in the callee procedure. In `whiskey()` this entails adding a downward constraint `a := a`$_n$, where `a`$_n$ is a specialized instance of `a` due to the inlining of `whiskey()`'s summary.

Furthermore, because the qualified model supports partial context sensitivity, an upward constraint is also added from the original to the specialized versions. This provides dataflow for context-insensitive derivations to proceed from the original to specialized instances. In `whiskey()` this entails adding the upward constraint `a`$_n$ `:= a`.

4.7 Modeling Heap Specialization

Section 2.5 introduced the general methodology behind the specialization of heap object and how such specialization can improve the accuracy of the pointer analysis result. Since different call paths to an allocation routine *may* create distinct heap objects, the fundamental goal is to speculatively model different heap object for each call path and thereby expose any differences.

The difficulty with heap specialization is balancing the gains that can be obtained if enough specialization is performed against the large cost of overspecializing. Algorithmically, it would be trivial to create a specialized heap object for every call path. If performed exhaustively, this might require the modeling of billions of objects. However, if one considers how programs are written, it is unlikely that there are even a few hundred truly distinct kinds of heap allocated objects. Thus, out of billions of possible paths a very small fraction should require specialization to obtain accurate results.

```
void main() {
    if( connection_machine ) { unlap1(-1); }
    if( doglobal ) { unlap2(-2); }
}

void unlap(int flag)
{
  int *left_queue = malloc3();
  int *right_queue = malloc4();
  int *center_queue = malloc5();

  for( i = 1 ; i <= cell_count ; i++ ) {
     left_queue[] = ...
     right_queue[] = ...
     center_queue[] = ...
  }
  ...
}
```

Figure 4.7 Example derived from 300.twolf. All the objects in this example have purely local effects; thus, specialization is not necessary.

4.7.1 Lossless specialization limits

As originally described in [25], the summarization process, buttressed by the qualified constraints and global variable isolation, provides three implicit mechanisms for limiting the amount of heap specialization. The first limit is that heap objects that do not interact at all with external data flow cannot benefit from specialization. Because they are not holding, such heap objects never become involved in the summarization process. Consider the example in Figure 4.7. In this example, arrays `left_queue`, `right_queue`, and `center_queue` are created only for local use and thus do not escape `unlap`. Therefore, further specialization of the arrays (differentiating those in $\texttt{unlap}_1$ and $\texttt{unlap}_2$) will unnecessarily incur overhead since each specialized version will be identical.

```
main() {
    int a = dummy (1);
    int b = dummy (2);
}

dummy (int m) {
    int *obj = malloc(sizeof int);
    *obj = m;
    return *obj;
}
```

Figure 4.8 Example demonstrating the effect of compaction for nonescaping objects.

Even if a heap object may hold an external reference, this is not sufficient to make it benefit from specialization. The second mechanism is that heap objects used purely as intermediate containers do not get included into summaries. Procedure `dummy` in Figure 4.8 creates an object of `int` type, assigns `m` to it, then returns its contents. From the perspective of caller `main`, only the net effect of callee `dummy` is important. Therefore, instead of specializing the entire callee, only its *summary* needs to be specialized. Since the object does not escape `dummy`, all its data flow can be represented in a compact form, i.e., `return m`. In this form, the same overall effect is achieved without specializing the heap object. From the perspective of `dummy`, the object can acquire either `1` or `2`. Specialization cannot change this fact.

While the exposure of a heap object to a caller is often an opportunity to improve accuracy through specialization when the object escapes only through a global variable, there is no benefit. This results for the same reason that global variables do not benefit from context sensitivity. Global variable isolation allows global variables to be correctly

```
typedef struct node { char n_type; struct node *n_next; } NODE;

NODE *newseg;

void main(void) {
   NODE *n1 = newnode1(1);
   NODE *n2 = newnode2(2);
   classify (n1->n_type);
}

NODE *newnode(int type) {
    NODE *nnode;
    if (newseg == NIL) { findmem(); }
    nnode = newseg;
    newseg = nnode->n_next;
    nnode->n_type = type;
    return (nnode);
}

void findmem() { newseg = calloc(1,ALLOCSIZE); ... }
```

Figure 4.9 Example derived from 130.li. In this example, since objects escape only through global variables, specialization does not aid accuracy.

excluded from summaries. This exclusion also ends specialization for heap objects that interact only with a global variable.

Consider the example in Figure 4.9. In this situation, the program uses a custom global allocation pool. The heap objects assigned into `n1` and `n2` are stored and fetched from a global allocation pool rooted at `newseg`. Specialization for $\texttt{newnode}_1$ and $\texttt{newnode}_2$ will not aid accuracy in a flow-insensitive analysis because global variable `newseg` will acquire both heap objects. Therefore, `n1` and `n2` end up pointing to both heap objects.

4.7.2 Lossy specialization limits

By eliminating fruitless paths, these mechanisms significantly reduce the cost of performing heap specialization. However, this is still not always sufficient. Two additional, potentially lossy, mechanisms exist to further reduce overhead. The ability to mark objects for context insensitive treatment applies to specialized heap objects as well. At any point during the specialization process, a heap object can be marked CI, thereby ending its specialization at that point in the call path (yet retaining any benefits from specialization that has taken place so far). Fulcra supports a option which can be set to limit the number of times a heap object is specialized. Once the limit is reached, the subsequent specialization attempts are marked CI and will no longer be included into summaries. While very useful for experimentation, this is a manual setting.

The other mechanism automatically recombines specialized heap objects for which specialization appears to have failed (i.e., the objects are nearly identical). At predefined intervals in the analysis process, this mechanism performs a quick partial comparison of existing heap objects. If two are roughly 95% similar then they are recombined. While still expensive, this mechanism helps reign back in the cost of fruitless specialization attempts.

The need for heap recombination was particularly noticed when heap specialization was attempted in the absence of field sensitivity and when the compiler performed large amounts of real inlining. Select data is shown in Table 4.4. Performing heap sensitive

Table 4.4 Excessive heap specialization occurs when *132.ijpeg* and *jpegenc* are analyzed field-insensitively and can occur in *176.gcc* when the amount of inlining in increased. The following shows both the cost of such excessive specialization on analysis time and how heap recollapse helps reign in these costs.

Benchmark	Config	without	with recollapse
132.ijpeg	hk, inline	INF	96
jpegenc	hk, inline	INF	112
176.gcc	h1, noinline	126	55
176.gcc	h1, inline	130	56
176.gcc	h1, moreinline	INF	715
176.gcc	hk, noinline	661	348
176.gcc	hk, inline	INF	342
176.gcc	hk, moreinline	INF	865

field-insensitive analysis can result in a large number of heap objects that differ little because the lack of field sensitivity is corrupting the result. Without heap recombination, *132.ijpeg* and *jpegenc* would not complete, whereas with it they required only a couple of minutes. Similar trends are shown for *176.gcc*. Inlining performed by the compiler effectively performs a part of a context sensitive analysis using summaries composed of entire procedures. This presents a problem for heap specialization because such whole scale replication bypasses the summarization's ability to prune away fruitless heap specialization. Heap recombination provides a mechanism to undo, from the perspective of the pointer analysis, some of the specialization performed by the compiler.

4.8 Key Algorithm Pseudo-Code

To close the discussion on context sensitivity, the following presents pseudo-code and derivation rules for the primary components. The algorithm for the fast two-phase

```
DO
    Top-down performed first and iteratively
    to obtain a solution preview and discover call graph
    DO
        delta = get-new-constraints(program)
        solve-constraints(delta)
        change = call-graph-update(program)
    WHILE (change)

    Bottom-up phase
    sorted-fn = topo-sort(program.callgraph)

    FOR (each fn in sorted-fn, last-to-first)
        create-and-apply-summaries(fn)
        delta = get-new-constraints(fn)
        solve-constraints(delta)
    ENDFOR
    change = call-graph-update(program)
WHILE (change)
```

Figure 4.10 Pseudo-code for the modified two-phase analysis process leveraging solution previews.

process is described first, followed by the algorithms central to procedure summarization. When combined with the qualified constraint derivation algorithms presented earlier, a complete context sensitive pointer analysis system can be constructed.

4.8.1 Fast two-phase algorithm

Pseudo-code for the two-phase pointer analysis process using solution previews is shown in Figure 4.10. It consists of two major parts. The first iterates between extending the solution preview and call graph discovery. The second performs the bottom-up summarization step. The routine get-new-constraints returns a list of any constraints that have yet to be processed. Routine solve-constraints applies the appropriate derivation rules to the provided constraints in a work list driven fashion. Finally, the routine create-and-apply-summaries creates summaries for any callees for which summaries have yet to be made and inlines specialized copies of each callee's summary.

4.8.2 Holding, opacity, and criticality

Figure 4.11 lists extended derivation rules for holding, opacity, and criticality. The rules include procedure returns and the properties *propagates*, *propagates-ret*, and *holding-esc*. The properties *propagates* and *propagates-ret* mark variables whose contents might flow into externally visible variables (a variables contents might propagate outside the procedure). The property *propagates-ret* specifically marks variables whose contents might exit via a procedure return statement. The check return-valid(v) returns true only if it is valid for a reference to variable `v` to return from the procedure. This is typically true only for heap objects. The property *holding-esc* is similar to *holding* except that it exists solely for those nodes holding references to opaque variables.

4.8.3 Summarization

Figure 4.12 shows the top-level algorithm for performing procedural summarization. A node n in the constraint graph is permanent, perm(n), if it will be included into the summary. All formal parameters, the return, and opaque variables are permanent by default. All critical constraints are enqueued into a work list for backtracing. Backtracing is attempted for each constraint edge in the work list along both their sources and destinations. Once backtracing has been attempted, if the source and destination

$$\frac{\mathit{param}(u) \vee \mathit{opaque}(u)}{\mathit{holding}(u)}$$

$$\frac{\mathit{return}(u)}{\mathit{propagates\text{-}ret}(u)}$$

$$\frac{\texttt{u:=v} \quad \mathit{propagates}(u)}{\mathit{propagates}(v)}$$

$$\frac{\texttt{u:=v} \quad \mathit{propagates\text{-}ret}(u)}{\mathit{propagates\text{-}ret}(v)}$$

$$\frac{\texttt{u:=v} \quad \mathit{holding}(v)}{\mathit{holding}(u)}$$

$$\frac{\texttt{u:=*v} \quad \mathit{holding}(v)}{\mathit{holding}(u)}$$

$$\frac{\texttt{*u:=v} \quad \mathit{holding}(u)}{\mathit{propagates}(v)}$$

$$\frac{\texttt{u:=\&v} \quad \mathit{propagates}(u)}{\mathit{opaque}(v)}$$

$$\frac{\texttt{u:=\&v} \quad \mathit{propagates\text{-}ret}(u) \quad \mathit{return\text{-}valid}(v)}{\mathit{opaque}(v)}$$

$$\frac{\texttt{u:=\&v} \quad \mathit{opaque}(v)}{\mathit{holding\text{-}esc}(u)}$$

(a)

$$\frac{*\, u := v \quad \mathit{holding}(u) \quad \mathit{holding}(v)}{\mathit{critical}(*u := v)}$$

$$\frac{*\, u := v \quad \mathit{holding}(u) \quad \mathit{holding\text{-}esc}(v)}{\mathit{critical}(*u := v)}$$

$$\frac{u := v \quad \mathit{opaque}(u) \quad \mathit{holding}(v)}{\mathit{critical}(u := v)}$$

$$\frac{u := *v \quad \mathit{opaque}(u) \quad \mathit{holding}(v)}{\mathit{critical}(u := *v)}$$

$$\frac{u := \&v \quad \mathit{opaque}(u)}{\mathit{critical}(u := \&v)}$$

(b)

Figure 4.11 Extended derivation rules for (a) holding, opacity, and (b) criticality. Though not explicitly shown, only horizontal constraints should be considered.

```
summarize-constraint-graph(constraint-graph cg)
    determine hold, opacity, criticality, etc..
    Keep nodes for inputs and outputs
    FORALL (nodes n in cg)
        IF (param(n) OR return (n) OR opaque(n))
            perm(n)
        ENDIF
    ENDFOR
    Enqueue all critical constraints
    FORALL (edges e in cg)
        IF (critical(e))
            queue-process(e)
        ENDIF
    ENDFOR
    Attempt to backtrace along source and dest of constraints
    FORALL (queue-process(e))
        backtrace(e,source-node(e));
        backtrace(e,dest-node(e));
        Add constraint to summary once source and dest have been selected
        IF (perm(source-node(e)) AND perm(dest-node(e)))
            add-to-summary(e);
        ENDIF
    ENDFOR
END-summarize-constraint-graph
```

Figure 4.12 Top-level algorithm for procedural summarization.

nodes of the constraint are both permanent no further backtracing was possible and the constraint is added to the summary.

The backtracing algorithm, in Figure 4.13, attempts to compact the edges along its source or destination node n. If done blindly, the backtracing can be overzealous in compacting away constraints in an attempt to remove redundant nodes before determining that a node must be retained due to some subsequently visited, uncompactable constraint. To avoid many poor backtracing choices, the algorithm first calls compact-effect to see if the constraint edge can be compacted along all inputs to n. If all inputs cannot be compacted, no input is compacted. The second call to compact-effect either compacts all input edges to n or, if one or more fails compaction, compacts none but instead adds the appropriate constraints to the work list for further backtracing.

```
backtrace(edge e1, node n)
    IF (perm(n))
        return
    ENDIF
    Test if the constraint can be compacted along all inputs
    If not, keep the node for all constraints
    FORALL(e2 inputs of n)
        w = source-node(e2)
        IF (!hold(w) OR !HZ(e2))
            skip
        ENDIF
        IF (compact-effect(e1,e2,n,TESTMODE) == NOCOMPACT)
            perm(n);
        ENDIF
    ENDFOR

    Perform the constraint compaction and enqueue relevant constraints
    FORALL(e2 inputs of n)
        w = source-node(e2)
        IF (!hold(w) OR !HZ(e2))
            skip
        ENDIF
        compact-effect(e1,e2,n,REALMODE)
    ENDFOR
END-backtrace
```

Figure 4.13 Algorithm for backtracing a constraint edge along either a source or destination node.

```
compact(edge e1, edge e2, node n, mode)
    (enew, ret-effect, ret-compact) = compact-core(e1,e2)
    IF (ret-effect == HASEFFECT)
        IF (perm(n) OR ret-compact == NOCOMPACT)
            Constraints cannot be compacted, enqueue input constraint
            IF (mode == REALMODE)
                queue-process(e2)
            ENDIF
            return NOCOMPACT
        ELSE
            IF (mode == REALMODE)
            Constraints can be compacted, enqueue resultant constraint(s)
                queue-process(enew)
            ENDIF
            return COMPACT
        ENDIF
    ELSE
        return NOEFFECT
    ENDIF
END-compact
```

Figure 4.14 Top-level constraint compact algorithm.

$$\frac{\texttt{u:=v} \quad \texttt{v:=w}}{e_{new} : \texttt{u:=w} \quad \text{HASEFFECT}}$$

$$\frac{\texttt{u:=v} \quad \texttt{v:=*w}}{e_{new} : \texttt{u:=*w} \quad \text{HASEFFECT}}$$

$$\frac{\texttt{u:=*v} \quad \texttt{v:=w}}{e_{new} : \texttt{u:=*w} \quad \text{HASEFFECT}}$$

$$\frac{\texttt{u:=*v} \quad \texttt{v:=*w}}{\text{NOCOMPACT} \quad \text{HASEFFECT}}$$

$$\frac{\texttt{*u:=v} \quad \texttt{v:=w}}{e_{new} : \texttt{*u:=w} \quad \text{HASEFFECT}}$$

$$\frac{\texttt{*u:=v} \quad \texttt{v:=*w}}{\text{NOCOMPACT} \quad \text{HASEFFECT}}$$

$$\frac{\texttt{*u:=v} \quad \texttt{v:=\&w}}{\text{NOCOMPACT} \quad \text{HASEFFECT}}$$

$$\frac{\texttt{*v:=u} \quad \texttt{v:=w}}{e_{new} : \texttt{*w:=u} \quad \text{HASEFFECT}}$$

$$\frac{\texttt{*v:=u} \quad \texttt{v:=*w}}{\text{NOCOMPACT} \quad \text{HASEFFECT}}$$

Figure 4.15 Constraint compaction rules for *compact-core()*.

The compaction of two constraints, shown in Figure 4.14, consists mostly of the core compaction rules. The rest of its function is to return NOCOMPACT and enqueue the input edge if the constraints could not be compacted together; otherwise, it returns COMPACT and enqueues the new, compacted constraint.

Figure 4.15 lists the compaction rules for the routine *compact-core()* called from *compact()* from Figure 4.14. The state HASEFFECT simply means that the two constraints represent some form of data flow. The state NOCOMPACT means that the constraints could not be further reduced.

5. FIELD SENSITIVITY

For the last several decades, many pointer analyses with significant variation in accuracy and scalability [6, 7, 13, 16, 17, 19–21] have been proposed for the C language. The C language offers few type safety guarantees giving C programmers extensive latitude in their use (and abuse) of types. Due to this lack of standard, formal type semantics, C pointer analysis frameworks are often built upon different sets of semantic assumptions. The choice of abstraction made by the analysis designer greatly affects the safety of the resultant pointer information and the efficiency of obtaining the result.

Type unsafety becomes a particular concern when pointer analysis uses field information to provide better resolution into the usage of aggregate data structures. Conventional wisdom dictates that field sensitivity yields a significant improvement in accuracy for a wide range of programs. On the other hand, the type unsafety of the C language complicates the design of a field-sensitive pointer analysis.

Section 5.1 discusses the issues that an analysis designer needs to consider when designing a safe field-sensitive pointer analysis for C. In particular, it discusses the need to select a model that accommodates the common instances of type unsafeness while not descending to so low a level of abstraction as to render results useless. It poses a practical set of requirements on valid program execution that most bug-free programs satisfy. Based on this motivation, Section 5.2 presents a core model for field sensitivity that is flexible enough to accommodate type unsafe programs.

Further compounding the situation, field sensitivity complicates the process of obtaining a context-sensitive analysis result. In particular, obtaining compact procedure summaries require more complicated derivation rules, are less apt to compact well, and can, in fact, explode in size if the algorithms are not designed well. Section 5.3 discusses the complications of designing a jointly context-sensitive field-sensitive framework and presents solutions based on the framework described in Chapter 4.

Finally, Section 5.4 presents useful simplifications and approximations designed to provide more efficient constraint resolution and Section 5.5 evaluates these techniques.

5.1 Field Sensitivity and Safety

A pointer analysis is considered safe if it computes pointer information that accounts for all possible executions of the input program. Unlike that designed for type-safe languages such as Java, the pointer analysis designed for the C language can assume only token restraints have been enforced on the manipulation of typed objects. In such cases, to be as safe as possible, the pointer analysis must minimize its reliance on unsafe assumptions about type usage. This issue becomes a particular concern when designing a field-sensitive pointer analysis.

Using two examples motivated from real programs, we demonstrate that (1) to obtain any useful pointer analysis results some restriction must be imposed on what accounts for a possible execution of the input program and (2) overrestriction must be avoided to increase the safe applicability of the pointer analysis.

The first example, motivated by *099.go*, demonstrates how an unrestricted model can dilute analysis results and make them useless. The second example, motivated by `memcpy`, a widely used routine in the C standard library, demonstrates that an overly restricted model could sometimes miss important realizable data flow. Consider the following code[1]:

Example 1:

```
void danio(int i) {
   int a; int b[1]; int c;
   c := b[i];
}
void otto() {
   int d;
   danio(...);
}
```

For this example, it is impossible to provide complete safety and useful analysis information at the same time. Assuming that all local variables are allocated on the call stack, unless a tight range of `i` can be proven, the expression `b[i]` could alias to `a`, `b`, `c`, `i` or even to `d`. While nothing can prevent a programmer from engineering an access

[1]Note that all examples will assume a typical 32-bit machine having 32-bit pointers and integers. Nevertheless, the methods presented in this paper work for other pointer and integer sizes.

outside of `b`, in reality such data flow is most likely a bug. A similar situation occurs in the SPECint95 benchmark *099.go* where the expression `g:=board[mvs[msptr-1]]` in `play_safe()` evaluates to `g:=board[-10]`. Since `board` is a global array and `g` is later used as an index, a different global data layout can cause a segmentation fault. Necula et al. [35] reported that similar situations are observed in other SPEC benchmarks.

On the other hand, a low-level modeling of data flow is sometimes unavoidable to capture all realizable data flow. Consider the following code:

Example 2:

```
int x;
void pleco() {
    struct st_t {int f; int *ptr;} s1, s2;
    s2.ptr := &x;
    mycopy((char*)&s1, (char*)&s2, 8);
}
void mycopy(char *dst, char *src, int n) {
   for (i:=0, j:=0; i<n; i++, j++)
     dst[i] := src[j];
}
```

Procedure `mycopy()` performs a simple memory copy from the object pointed to by `src` into the object pointed to by `dst`. In this case, part or all of structure `s2` will be copied into `s1`. Based on an unsafe assumption about types, a pointer analysis might ignore the passage of `s1` into `mycopy()` because of the type cast to `char*`. Alternatively, it might ignore the assignment `dst[i] := src[j]` because it is a character copy. Omission of either leads to the absence of the copying and the points-to relationship between `s1.ptr` and `x`. To compound the problem, the pointer analysis may instead model `dst[i] := src[j]` as a block copy action. However, this requires an extra analysis on the loop to determine the correlation between `i` and `j`; without such analysis, the pointer analysis should conclude that any offset in `dst` may obtain any address stored within `src`.

Therefore, pointer analysis must faithfully adhere to a useful model, which is general enough to provide safe results for the vast majority of programs, yet restricted enough to allow for useful and timely results. The solution requires consideration of a number of C features:

1. Casting. In C, types may change without the change in raw data. Pointer analysis will frequently miss real data flow if it is reliant upon tests for "compatible" types, where data flow involving "incompatible" types is assumed false and filtered away.

2. Unions. While not preferred coding practice, data may flow into one field of a union and out another. Omission of this path may lead to missing data flow.

3. Structure-structure block copy. When one multibyte structure is copied into another, the compiler inserts a block copy. It could be modeled incorrectly as a sequence of copies, one for each field in the data structure. Consider the following code:

Example 3:

```
struct st_t1 {int *a; int *b;} s1, s2;

struct st_t2 {short c,d,e,f;} *p1, *p2;

p1 := (struct st_t2 *)&s1;

p2 := (struct st_t2 *)&s2;

*p1 := *p2;
```

Since `st_t1` and `st_t2` are the same size, `st_t2` could be used to copy `s2` to `s1`. If fields are copied individually, the framework may miss the copy of `a` and `b` because neither is covered by `c`, `d`, `e`, or `f`. A safer option is to model the statement as a block copy of 8 bytes, accurately modeling the flow of pointer information regardless of type casting.

4. Pointer Arithmetic. Taking the address of an arrow operator can lead to implicit pointer arithmetic. For example:

Example 4:

```
struct st_t {int *a; int *b;

              int *c; int *d; } *ptr, s;

ptr := &s;

loop() {  ptr := &ptr->b; }
```

Assignment `ptr := &ptr->b` is equivalent to `ptr := ptr + 4` and the loop around it means that `ptr` may point to any field offset, `s.a`, `s.b`, `s.c`, or `s.d`. Similarly, a programmer may explicitly calculate an offset into a multibyte data structure. In such cases, field accesses occur without explicitly naming fields.

5. Heap. It is not always possible to statically know the size much less the type of a particular dynamically allocated chunk of memory. This introduces additional unknowns.

6. Arrays. As depicted by the first example, array bounds are in general difficult to determine. In some cases, even the indexing pattern may be unknown due to the substantial interaction with other data structures, as in Example 2.

Most of the above challenges are in violation of various C standards. While it is valid to heavily restrict the accepted input programs, doing so would substantially limit the framework's applicability and, for example, would eliminate many programs in a benchmark suite like SPECint.

In this regard, our goal is two-fold: (1) To present a useful model that is known to be conservative with respect to commonly used C behavior. This model correctly covers all of the C features listed above. (2) To present and evaluate a practical field-sensitivity framework that supports the model. Our model is not without limitations. We have settled on three important restrictions on input programs:

1. **No arithmetic calculation can be used to go from one distinct object to another.** For example, a program's correctness should not be reliant upon `*(&a + 8)` aliasing with `b`, as in Example 1. Likewise, the program must not assume that one heap allocation can be reached through the arithmetic manipulation of

another allocation's pointer. Since static bounding of indices is often unfruitful, relaxing this qualification in the presence of arrays or pointer arithmetic would render results hopelessly conservative. This does not preclude pointer arithmetic within a given structure or array, however.

2. **Valid pointers cannot be materialized from nonpointers.** A pointer value may be copied in chunks as in Example 2; a pointer value may not, however, be created spontaneously from a collection of constants or other data flow not rooted in an address operation. While beyond the scope of this dissertation, this restriction could be relaxed as long as references to fixed addresses can be identified, This relaxation would be useful for analyzing non-user programs. However, complete removal of this restriction would require all numerical values to be treated as pointer values accessing objects of unkown size. The natural conservatism of the analysis process could then lead to a number of false dereferences to non-pointers which would likely compound and degenerate the usefulness of the results.

3. **The whole program is available.** As with the second restriction, this can be relaxed, in which case the problem becomes one of deriving or providing a useful summary of the possible effects of missing program componenets. While feasible this again lies outside the scope of this dissertation.

Given these restrictions, the first focus is to present a general model for field sensitivity capable of safely modeling all C programs obeying the above limitations.

5.2 Constraint Specification

The flow of pointer values in a program is modeled as the interaction of five kinds of assignments with memory objects. The following subsection details the representation of objects and assignment constraints and the derivation rules used to derive further constraints. Constraint resolution involves construction of the initial constraint graph and application of the derivation rules until convergence. A short example is included. Important implementation issues are covered in Section 5.4.

5.2.1 Object model

The C language does not enforce a strong type system and C programmers often neglect even the most salutary typing conventions; thus, reliance on type compatibility in analysis can easily lead to incomplete, and therefore erroneous, conclusions. Our safe analysis, on the other hand, models objects from a machine perspective. To this end, all program objects, regardless of type, are represented as a pair, (id, z), where id identifies an object and z its size. Every memory location used by a program can be mapped to an id. There is a one-to-one mapping between *named*, statically allocated, objects and an id while a single id may represent many *anonymous*, dynamically allocated, objects.

Since the compiler explicitly allocates space for named objects, it has full knowledge of the size of named objects. Thus, accesses beyond an object's bounds can be considered false, allowing the filtering out of their data flow. However, it is not always possible to statically determine the size of anonymous objects. One might erroneously set the size of these objects to the maximal declared type size, but, in C, the largest allocated object size may be larger than the maximum *declared* type size. Consider the following expression, modeled after the *176.gcc* `rtx` data structure:

```
struct data_t { int *a, int *b };
struct base_t { int *ptr, data_t end[0] } *st_ptr;
st_ptr = malloc(sizeof(base_t) + sizeof(data_t));
z := ((data_t*)(&st_ptr->end))->b;
```

The expression `&st_ptr->end` is used to mark the end of the data structure. The cast to `data_t*` allows access to fields outside of `base_t*`. The maximum declared type size is 8 bytes while the maximum allocated object is 12 bytes.

Since it is not always possible to statically determine the size or intended type of particular anonymous objects, they must be modeled as having unbounded size (i.e., not setting an a priori limit on an object's size). Section 5.4.1 discusses how this is approximated in our implementation.

5.2.2 Derivation rules

The model used to represent pointer data flow in a program consists of the following five assignment constraints. All offsets and sizes are byte oriented.

Plain	$u_t \overset{z}{:=} v_s$	Assign the z bytes starting at v offset s to u offset t
Load	$u_t \overset{z}{:=} *v_s$	Assign z bytes from the locations pointed-to by v offset s to u offset t
Store	$*u_t \overset{z}{:=} v_s$	Assign z bytes from v offset s to the locations pointed-to by u offset t
Addr	$u_t := \&v_s$	Assign the address of v offset s to u offset t
Skew	$u_t := v_s + k$	Add k to the offset of each location pointed-to by v offset s and assign these resultant addresses to u offset t

The constraint solution algorithm works by propagating address constraints along assignments, creating new address constraints by adding to existing ones, and deriving new plain assignments by combining address and load or store constraints. The core solution process is similar to that used in Foster et al. [20]. The proposed field sensitivity model adds the complexity of determining if two constraints overlap in any meaningful way and the materialization of this overlap in any derived constraints.

The constraint solution process involves the following four derivation rules:

Address Propagation

$$\frac{u_{t1} \overset{z}{:=} v_{s1} \quad v_{t2} := \&w_{s2} \quad s1 <= t2 < s1 + z}{u_{t1+(t2-s1)} := \&w_{s2}}$$

Address Arithmetic

$$\frac{u_{t1} := v_{s1} + k \quad v_{t2} := \&w_{s2} \quad t2 = s1 \quad valid(w, s2 + k)}{u_{t1} := \&w_{s2+k}}$$

Load From Address

$$\frac{u_{t1} \overset{z}{:=} *v_{s1} \quad v_{t2} := \&w_{s2} \quad t2 = s1}{u_{t1} \overset{z}{:=} w_{s2}}$$

Store To Address

$$\frac{*\, u_{t1} \overset{z}{:=} v_{s1} \quad u_{t2} := \&w_{s2} \quad t2 = t1}{w_{s2} \overset{z}{:=} v_{s1}}$$

The function *valid(object, offset)* is true if the offset is less than z, the object's allocated size. It will always return true for dynamically allocated objects because their size is unbounded.

A visual model for the address propagation derivation is shown in Figure 5.1. The address of offset $s2$ of `W` is being assigned into offset $t2$ of `V`. The z consecutive bytes starting at offset $s1$ of `V` are being copied as a block into a region of `U` starting at $t1$. The address will flow through the block copy only if $t2$ is within the range being copied. The expression $t2 - s1$ is the relative offset of the address within the block being copied, thus adding it to $t1$ results in the target offset in `U`. Also note that partially overlapped assignments do not introduce problems for the model because an access to an address is named only by its starting offset. For example, writes of `&a` into offset 4 of object `X` and

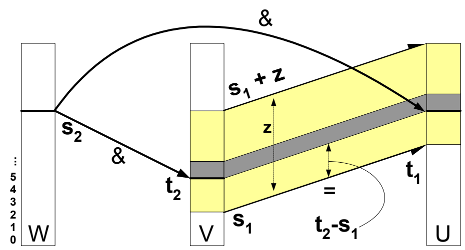


Figure 5.1 Illustration of the address propagation derivation.

`&b` into offset 5 of object `X` do not conflict. Accesses to offsets 4 and 5 will obtain `&a` and `&b`, respectively.

5.2.3 Example

Figure 5.2(a) illustrates a constraint graph for the source in Example 2 from Section 5.1. Objects are marked with the id and size pair from Section 5.2.1. Constraints are annotated in the form $k(s,t)$. Each assignment is of kind k, one of: address `&`, skew `+`, plain `=`, load `=*`, or store `*=`. The pair s and t are the source and target offsets, respectively. Note that the modeling of the C expression `dst[] := src[]` required the use of two constraints and the temporary `tmp`.

Figure 5.2(b) depicts the result of applying the derivation rules from Section 5.2.2 on the initial constraint graph. For example, the arithmetic cycle `+1(0,0)` at `src` in combination with the address constraint from `s2` to `src` produces seven additional address constraints. For visual clarity, ellipses are used when constraints spanning a range are derived; e.g., `&(0,0)...&(0,7)` represents all target offsets from `0` to `7`, inclusive. One result of the solution process is that any offset of `s1` may contain a reference to `x`.

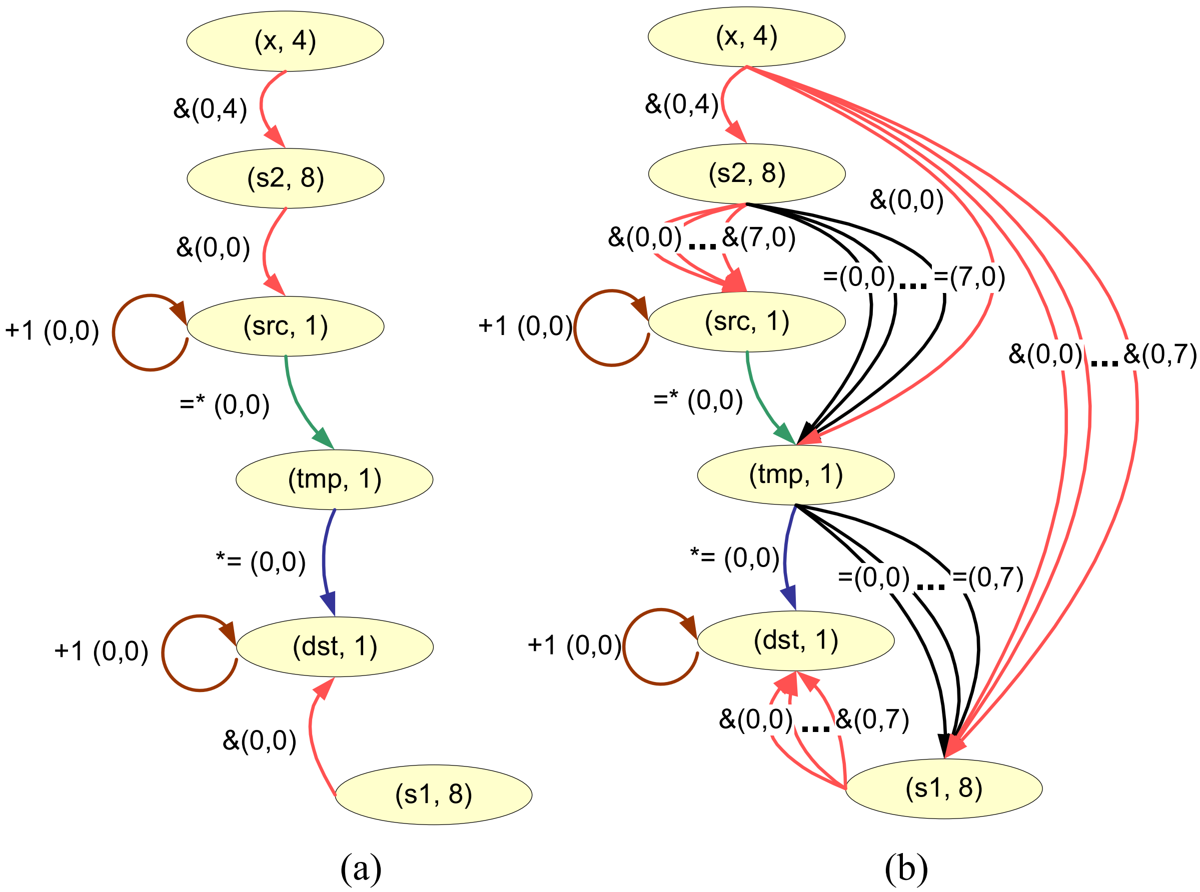


Figure 5.2 Example: (a) Initial constraint graph for code of Example 2, and (b) solution to constraints.

The example demonstrates the derivation of a safe result for a complex scenario. However, it also shows that the complete enumeration of constraints can lead to a large number of constraints. This and other efficiency issues will be addressed in Section 5.4.

5.2.4 Safety issues

This section revisits the safety issues outlined in Section 5.1 to show how the proposed safe field sensitivity system accommodates each of them.

1. Casting. While types are used by the compiler for the allocation of named objects and for address calculations, they are not used by the pointer analysis and, thus, pose no problems.

2. Unions. Physical offsets are used for accessing fields, thus unions are conservatively modeled.

3. Structure-Structure Block Copy. Multiple-byte assignments are modeled as such, using the size of assignments (Example 2 and Figure 5.2).

4. Pointer Arithmetic. The skew constraint is used to explicitly model all address arithmetic. This may result in cyclic computation of data structure offsets intra- and interprocedurally. The skew k in the skew constraint is derived from source code expressions. If the source expression can be reduced to the addition of an integer value, k is set to this value; otherwise, k is set to a conservative size of 1, or, in the case of arrays, element size.

5. Heap. Heap objects are modeled with an unbounded size. [2] Since type information is not used, unknown type and layout of heap objects do not pose a problem.

6. Arrays. Array are largely covered under the pointer arithmetic case above. However, for expressions of the form `array[i]` where `i` is unknown, an arithmetic cycle with a skew k, where k is the array's element size, must be added explicitly. Such cycles can be seen on `dst` and `src` in Figure 5.2(a).

[2] This could be made less conservative by using the size when it can be definitively determined from the call to `malloc()`. See Section 5.4 for more details.

Mike(p) {

$\quad a_0 \overset{16}{:=} p_0;$

$\quad b_0 \overset{16}{:=} a_1 6;$

}

Figure 5.3 Example for which field-insensitive computation of the holding property results in a conservative result.

5.3 Interaction with Context Sensitivity

Chapter 4 detailed a qualified constraint approach to obtaining a context sensitive pointer analysis. The bottom-up phase of the analysis computed procedure summaries to permit the hoisting of side-effects from callees to callers. There are two major steps to the summarization process; in particular, the properties of holding and opacity are computed for constraint graph nodes and critical constraint assignments are located to seed a backtracing process. The existence of field offsets can complicate both steps and, if performed naïvely, may result in summary size explosion.

5.3.1 Holding, opacity, and criticality

Within a field insensitive environment, a *variable* is *holding* if it might contain a reference to a memory location external to the procedure, and a *variable* is *opaque* if its contribution to actual data flow is uncertain without specific knowledge about the procedure's calling context. Both were defined in terms of an entire variable. However, within a field sensitive environment, offsets must be considered to avoid an overapproximation of holding and opacity properties. Consider the code example in Figure 5.3. A field insensitive determination of holding and opacity will lead to `b` holding. However, when offsets and assignment sizes are accounted for there is, in fact, no data flow from the parameter to `b`.

This presents two options for determining these properties. First, the holding and opacity derivation rules can be extended to compute the intervals over which the property exists. In Figure 5.3, `a` would hold over the interval $[0-15]$ and thus `b` could not obtain the holding property from `a` interval $[16-31]$. The key downside to computing the exact result is that it makes a previously simple computation much more complex. Furthermore, this produces a more complicated set of information for the backtracing process to consume.

The second option is to simply compute the properties field-insensitively and live with the overapproximation. If overapproximation translated directly into larger summaries, a field-sensitive computation would be the best choice because keeping summary sizes is a critical goal. However, even if provided an overapproximation of holding, opaqueness, and criticality, the backtracing process, which is field sensitive, naturally prunes out false data flow. For this reason, the determination of holding and opacity and therefore constraint criticality are based on the field insensitive derivation rules and algorithms presented in Section 4.8.

5.3.2 Backtracing and summarization

The backtracing process for the summarization within a field-sensitive environment is similar in concept to the field insensitive process described in Section 4.8. In fact, only two algorithms require modification. First, the backtrace algorithm in Figure 4.13 has three additional lines of code. Second, the core compact rules in Figure 4.15 are extended to become field sensitive.

Figure 5.4 shows the new backtracing routine. The additions to the field-sensitive version are shown in bold. The condition prevents the backtracing algorithm from attempting to explicitly expand arithmetic cycles which can result in explosive growth in summary size.

Figures 5.5 – 5.8 present the new compaction rules when modified to accommodate field sensitivity based on copy, skew, load, and store constraints, respectively. The BOUND(s, m, e) function simply checks to see if $s <= m <= e$. While extensive, many have calculations in common. There are two forms of the BOUND checks, BOUND($t2, s1, t2 + z2$) and BOUND($s1, t2, s1 + z1$), along with corresponding relative offset calculations, $s2 + (s1 - t2)$ and $t1 + (t2 - s1)$.

The derivation rules for 5.5(d) and 5.8(b) are the only two compaction steps that result in two constraints instead of one. For 5.5(d), `u:=v` and `v:=*w` are being compacted together resulting in `u:=*w`. However, if the source offset for `v` in `u:=v` is larger than the target offset for `v` in `v:=*w` then the source of `u:=*w` should be increased correspondingly. One might try to reflect this fact by resolving to $\texttt{u:=*w}_{s2+shift}$. However, this does not shift the source of `*w` in `u:=*w` but, instead, changes the field of `w` that gets dereferenced which is clearly not the goal. Instead, the objective is to shift the source of any assignment derived by `*w`. This effect can be represented by adding the shift value to any address

```
backtrace(edge e1, node n)
    IF (perm(n))
        return
    ENDIF
    Test if the constraint can be compacted along all inputs
    If not, keep the node for all constraints
    FORALL(e2 inputs of n)
        w = source-node(e2)
        IF (!hold(w) OR !HZ(e2))
            skip
        ENDIF
        Do not explicitly expand arithmetic cycles
        IF (skew-edge(e2) AND source-node(e2) == dest-node(e2))
            perm(n);
        ENDIF
        IF (compact-effect(e1,e2,n,TESTMODE) == NOCOMPACT)
            perm(n);
        ENDIF
    ENDFOR

    Perform the constraint compaction and enqueue relevant constraints
    FORALL(e2 inputs of n)
        w = source-node(e2)
        IF (!hold(w) OR !HZ(e2))
            skip
        ENDIF
        compact-effect(e1,e2,n,REALMODE)
    ENDFOR
END-backtrace
```

Figure 5.4 Algorithm for field-sensitive backtracing a constraint edge along either a source or destination node.

(a) $$\frac{\mathtt{u}_{t1} \overset{z1}{:=} \mathtt{v}_{s1} \quad \mathtt{v}_{t2} \overset{z2}{:=} \mathtt{w}_{s2} \quad \mathrm{BOUND}(t2, s1, t2+z2)}{e_{new} : \mathtt{u}_{t1} \overset{\min(z1, z2-(s1-t2))}{:=} \mathtt{w}_{s2+(s1-t2)} \quad \mathrm{HASEFFECT}}$$

(b) $$\frac{\mathtt{u}_{t1} \overset{z1}{:=} \mathtt{v}_{s1} \quad \mathtt{v}_{t2} \overset{z2}{:=} \mathtt{w}_{s2} \quad \mathrm{BOUND}(s1, t2, s1+z1)}{e_{new} : \mathtt{u}_{t1+(t2-s1)} \overset{\min(z2, z1-(t2-s1))}{:=} \mathtt{w}_{s2} \quad \mathrm{HASEFFECT}}$$

(c) $$\frac{\mathtt{u}_{t1} \overset{z1}{:=} \mathtt{v}_{s1} \quad \mathtt{v}_{t2} := \mathtt{w}_{s2} + k \quad \mathrm{BOUND}(s1, t2, s1+z1)}{e_{new} : \mathtt{u}_{t1+(t2-s1)} := \mathtt{w}_{s2} + k \quad \mathrm{HASEFFECT}}$$

(d) $$\frac{\mathtt{u}_{t1} \overset{z1}{:=} \mathtt{v}_{s1} \quad \mathtt{v}_{t2} \overset{z2}{:=} *\mathtt{w}_{s2} \quad \mathrm{BOUND}(t2, s1, t2+z2)}{e_{new} : \{\mathtt{u}_{t1} \overset{\min(z1, z2-(s1-t2))}{:=} *\mathtt{t}, \mathtt{t} := \mathtt{w}_{s2} + (s1-t2)\} \quad \mathrm{HASEFFECT}}$$

(e) $$\frac{\mathtt{u}_{t1} \overset{z1}{:=} \mathtt{v}_{s1} \quad \mathtt{v}_{t2} \overset{z2}{:=} *\mathtt{w}_{s2} \quad \mathrm{BOUND}(s1, t2, s1+z1)}{e_{new} : \mathtt{u}_{t1+(t2-s1)} \overset{\min(z2, z1-(t2-s1))}{:=} *\mathtt{w}_{s2} \quad \mathrm{HASEFFECT}}$$

Figure 5.5 Core field-sensitive copy constraint compaction rules.

(a) $$\frac{\mathtt{u}_{t1} := \mathtt{v}_{s1} + k1 \quad \mathtt{v}_{t2} \overset{z2}{:=} \mathtt{w}_{s2} \quad \mathrm{BOUND}(t2, s1, t2+z2)}{e_{new} : \mathtt{u}_{t1} := \mathtt{w}_{s2+(s1-t2)} + k1 \quad \mathrm{HASEFFECT}}$$

(b) $$\frac{\mathtt{u}_{t1} := \mathtt{v}_{s1} + k1 \quad \mathtt{v}_{t2} := \mathtt{w}_{s2} + k2 \quad (t1 == t2)}{e_{new} : \mathtt{u}_{t1} := \mathtt{w}_{s2} + (k1 + k2) \quad \mathrm{HASEFFECT}}$$

(c) $$\frac{\mathtt{u}_{t1} := \mathtt{v}_{s1} + k1 \quad \mathtt{v}_{t2} \overset{z2}{:=} *\mathtt{w}_{s2} \quad \mathrm{BOUND}(t2, s1, t2+z2)}{\mathrm{NOCOMPACT} \quad \mathrm{HASEFFECT}}$$

Figure 5.6 Core field-sensitive skew constraint compaction rules.

(a) $$\frac{\mathtt{u}_{t1} \overset{z1}{:=} *\mathtt{v}_{s1} \quad \mathtt{v}_{t2} \overset{z2}{:=} \mathtt{w}_{s2} \quad \mathrm{BOUND}(t2, s1, t2+z2)}{e_{new} : \mathtt{u}_{t1} \overset{\min(z1)}{:=} *\mathtt{w}_{s2+(s1-t2)} \quad \mathrm{HASEFFECT}}$$

(b) $$\frac{\mathtt{u}_{t1} \overset{z1}{:=} *\mathtt{v}_{s1} \quad \mathtt{v}_{t2} \overset{z2}{:=} \mathtt{w}_{s2} + k2 \quad (s1 == t2)}{\mathrm{NOCOMPACT} \quad \mathrm{HASEFFECT}}$$

(c) $$\frac{\mathtt{u}_{t1} \overset{z1}{:=} *\mathtt{v}_{s1} \quad \mathtt{v}_{t2} \overset{z2}{:=} *\mathtt{w}_{s2} \quad \mathrm{BOUND}(t2, s1, t2+z2)}{\mathrm{NOCOMPACT} \quad \mathrm{HASEFFECT}}$$

Figure 5.7 Core field-sensitive load constraint compaction rules.

(a) $$\frac{\texttt{*u}_{t1} \overset{z1}{:=} \texttt{v}_{s1} \quad \texttt{v}_{t2} \overset{z2}{:=} \texttt{w}_{s2} \quad \text{BOUND}(t2, s1, t2+z2)}{e_{new} : \texttt{*u}_{t1} \overset{\min(z1, z1-(s1-t2))}{:=} \texttt{w}_{s2+(s1-t2)} \quad \text{HASEFFECT}}$$

(b) $$\frac{\texttt{*u}_{t1} \overset{z1}{:=} \texttt{v}_{s1} \quad \texttt{v}_{t2} \overset{z2}{:=} \texttt{w}_{s2} \quad \text{BOUND}(s1, t2, s1+z1)}{e_{new} : \{\texttt{t} := \texttt{*u}_{t1} + (t2-s1), \texttt{*t} \overset{\min(z2, z1-(t2-s1))}{:=} \texttt{w}_{s2}\} \quad \text{HASEFFECT}}$$

(c) $$\frac{\texttt{*u}_{t1} \overset{z1}{:=} \texttt{v}_{s1} \quad \texttt{v}_{t2} := \texttt{w}_{s2} + k2 \quad \text{BOUND}(s1, t2, s1+z1)}{\text{NOCOMPACT} \quad \text{HASEFFECT}}$$

(d) $$\frac{\texttt{*u}_{t1} \overset{z1}{:=} \texttt{v}_{s1} \quad \texttt{v}_{t2} \overset{z2}{:=} \texttt{*w}_{s2} \quad (\text{BOUND}(s1, t2, s1+z1) \vee \text{BOUND}(t2, s1, t2+z2))}{\text{NOCOMPACT} \quad \text{HASEFFECT}}$$

(e) $$\frac{\texttt{*u}_{t1} \overset{z1}{:=} \texttt{v}_{s1} \quad \texttt{v}_{t2} := \texttt{\&w}_{s2} \quad (s1 == t2)}{\text{NOCOMPACT} \quad \text{HASEFFECT}}$$

(f) $$\frac{\texttt{*v}_{t1} \overset{z1}{:=} \texttt{u}_{s1} \quad \texttt{v}_{t2} \overset{z2}{:=} \texttt{w}_{s2} \quad \text{BOUND}(t2, s1, t2+z2)}{e_{new} : \texttt{*w}_{s2+(t1-t2)} \overset{z1}{:=} \texttt{u}_{s1} \quad \text{HASEFFECT}}$$

(g) $$\frac{\texttt{*v}_{t1} \overset{z1}{:=} \texttt{u}_{s1} \quad \texttt{v}_{t2} := \texttt{w}_{s2} + k2 \quad (t1 == t2)}{\text{NOCOMPACT} \quad \text{HASEFFECT}}$$

(h) $$\frac{\texttt{*v}_{t1} \overset{z1}{:=} \texttt{u}_{s1} \quad \texttt{v}_{t2} \overset{z2}{:=} \texttt{*w}_{s2} \quad \text{BOUND}(t2, s1, t2+z2)}{\text{NOCOMPACT} \quad \text{HASEFFECT}}$$

Figure 5.8 Core field-sensitive store constraint compaction rules.

incoming to `w`. The compaction step 5.5(d) results in the dereference `u:=*t` and a skew of the source `t:=w + k`. The reasoning for 5.8(b) is very similar.

5.4 Effective Implementation

The model described in Section 5.2 fully enumerates assignment constraints for every access to an object. While safe and high resolution, a direct implementation of this would be, at best, inefficient. This section addresses the modeling of anonymous objects with unbounded size and the compression of the constraints through the detection and use of strides.

5.4.1 Anonymous objects

As explained in Section 5.2.1, anonymous objects must be modeled as having unbounded size. An enumerative and exact approach to this would invalidate any termination guarantees. As an approximation, the representation of objects is extended to allow an object to wrap around onto itself. This is achieved through the addition of a modulus m to the state of an object. All accesses to an offset f are converted to accesses to $f \ \% \ m$. Figure 5.9 illustrates the effect where (a) shows a fully enumerated result and (b), (c), and (d) use moduli of 4, 3, and 2, respectively. The fully enumerated result in (a) reflects a traversal of 4-byte sized elements in `s` by `ptr`. With a modulus of 4 the representation preserves the pattern of traversal. By choosing a modulus of 3 the representation now conservatively indicates that `ptr` can point to any byte position. The

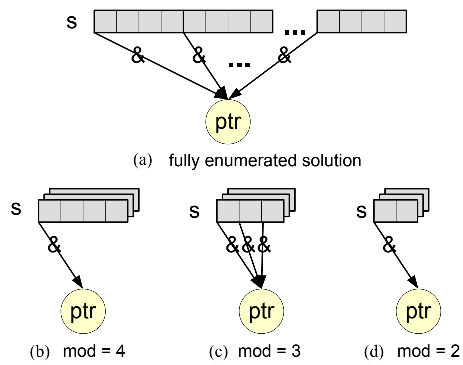


Figure 5.9 Effect of modulus on view of a multibyte structure.

choice of a modulus may impact the accuracy, but, since fields are never omitted (only combined), it never affects correctness. Selection of a particular modulus is covered in Section 5.4.3.

5.4.2 Modeling strides

With finite objects, the critical remaining inefficiency is the explicit enumeration of all patterned accesses. The following example will be used to illuminate the problem and the possible solutions:

```
struct {int a; short b,c; } st[3];
void foo() {
   for (i:=0, ptr:=(char*)&st; i<24; ptr++,i++) {}
   x := st[j].b;
}
```

Figure 5.10(a) depicts the resultant address constraints for this example. This very simple code sequence results in 24 address constraints into `ptr` and three into `x`. These

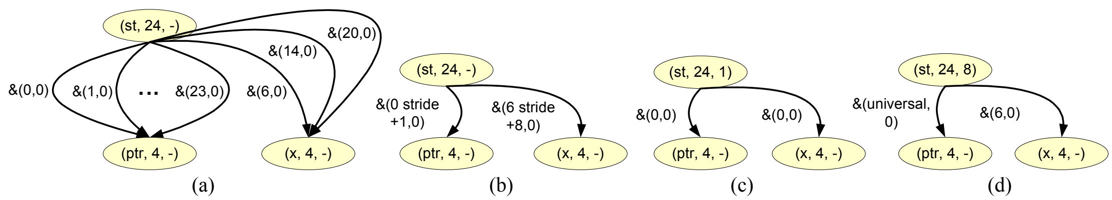


Figure 5.10 (a) Enumerated solution, (b) strides on constraints, (c) strides on objects, and (d) hybrid approach.

24 constraints correctly model the fact that `ptr` can point to any of the 24 byte positions of `st`. Full enumeration is not a scalable approach. A key simplification is the use of a stride to reduce the number of constraints. Note that the use of a stride improves efficiency, but is not required for correctness, as, by default, the model enumerates the constraints.

If it is known that a strided access exists, there are two mechanisms in our system through which this stride can be modeled. The first and most flexible option is to annotate the source and target offsets of constraints with stride information. This is shown in Figure 5.10(b) where the solution requires only two constraints, one with a stride of 1 and the other of 8.

While expressive, the use of strides on constraints can cause the analysis to become very slow. Referring back to Section 5.2.2, each derivation involves a check for overlap between the target of the defining constraint and the source of the consuming constraint. If overlap exists, the amount of overlap is used to generate the new constraint. The overlap calculation is one of the most frequent in the pointer analysis framework numbering in the 100s of millions for many SPECint2000 benchmarks, such as *255.vortex* and *254.gap*.

With arbitrary strides on constraints, this high frequency calculation entails obtaining the intersection of $\texttt{stride}_1 * i + \texttt{offset}_1$ and $\texttt{stride}_2 * j + \texttt{offset}_2$ in the form $\texttt{stride}_3 * k + \texttt{offset}_3$. This operation is further complicated by the access size z which may require the intersection to generate multiple constraints. For example, intersecting two size 2 assignments one with stride 4 and the other stride 5, results in $20 * i$ size 2, $20 * i + 5$ size 1, and $20 * i + 16$ size 1. The possibility of such constraint fragmentation along with the potential overhead of performing this operation millions of time makes arbitrary use of stride information on constraints unreasonably expensive as a sole solution.

A second, very efficient mechanism applies the stride directly to the object. This process leverages the same modulus extension used to make anonymous objects wrap onto themselves. Setting the modulus m of an object to the desired stride has the effect of applying the same stride to every constraint that interacts with the object. However, if multiple strides $s_1...s_n$ must be applied to the same object, the object obtains the greatest common denominator of $s_1...s_n$. In Figure 5.10(c), the mixed strides result in a stride of 1 being applied to `st`, reducing the accuracy of the accesses by `x`. While efficient, the mixing of strides can result in substantial loss of accuracy to large, multibyte data structures. In Figure 5.10(c), `x` is now assumed to point to any of the bytes positions in `st`.

A mixture of both approaches is used to avoid the individual weaknesses of each option. Strides are partitioned into *coarse-grain*, those with a stride greater than the size of a pointer, and *fine-grain*, those with a stride of pointer or subpointer size. Coarse-grain strides are applied to objects. While updating an object's modulus may require adjusting existing constraints, it does not change the derivation rules in Section 5.2.2 and does not adversely affect analysis time. Fine-grain accesses are reduced to the *universal* stride offset 0, stride 1, and are applied to the constraints. Fine-grain accesses become a localized loss of field sensitivity and intersections involving them are trivial to compute. The use of a universal stride on edges prevents fine-grained accesses from adversely affecting coarse-grained accesses. A result from this combined approach is shown in Figure 5.10(d). Section 5.5.1 will present an empirical justification of this hybrid approach.

5.4.3 Detecting strides

In order to apply any of the simplifications from Section 5.4.2, it is necessary to know a strided access exists, preferably before such an access has caused enumeration of constraints.

We apply two complementary approaches. The vast majority of all strided accesses result from intraprocedural cyclic computation. Cycle detection performed before the analysis process can find such cycles inexpensively. If found, the stride is computed as the greatest common denominator of the ks for all skew constraints involved in the cycle. All objects in the cycle are then marked with the stride. If, during the analysis process, an address constraint is derived with a left-hand targeting an object marked with a stride, the stride is applied in accordance with the rules presented in the previous section. This approach proactively applies strides before any enumeration takes place. At a high-level the goal is similar to Fähndrich et al. [36], though arithmetic cycles are not collapsed as in, just annotated.

Since this preemptive approach is intraprocedural, cycles that span multiple procedures will not be found. Conceptually, one could just extend the cycle detection to span the program-wide constraint graph. However, the expense of such a global check and the very infrequent occurrence of such cycles make this a heavy handed approach. Instead, a simple online (i.e., active during the analysis process) monitor looks for impending constraint explosion due to the enumeration of a strided access. If detected, the monitor approximates and applies the stride. Because the stride has been partially enumerated, the application of the stride involves collapsing these enumerated constraints.

The online monitor works as follows. Given a node u, the monitor computes n, the number of address constraints for which u is the source, and t, the number of unique targets of these constraints. The value of t may be less than n because a single target may point to multiple offsets of u. If u is not accessed in a cyclic fashion, the ratio n/t should be very close to 1. If the monitor obtains a ratio greater than 1.5, a stride is calculated for u and applied appropriately. The operation of the monitor can become expensive if it acts frequently; its frequency of operation, however, can easily be controlled by leveraging the fact that a ratio n/t cannot surpass 1.5 until $1.5t - n$ more address constraints have been added to u.

In practice, it is relatively rare for the monitor to discover cycles and strides not covered by preemptive cycle detection. Such cases are often the result of false data flow. Figure 5.11 shows an interesting example from *134.perl*.

Part 1 of Figure 5.11 shows a hash table procedure `hfetch()`, meant for strings, being used by `stabent()` to hash nonstrings. Procedure `hfetch()` returns a pointer to global variable `str_undef` as a not-found condition. However, it would be incorrect for `str_undef` to be returned by `stabent()` so it checks for this and returns `Nullstab` in its place. The pointer analysis sees a possible return of `str_undef` because it does not consider control flow or control conditions. Thus, the global `defstab` appears to point-to `str_undef`.

In part 2, `str_free()` contains a similar check for `str_undef` to avoid freeing global variable `str_undef`. Again, the analysis sees a possible flow of `str_undef` into the free list `froot` and, subsequently, a possible return through `str_new()`.

Parts 1 and 2 lead to part 3 creating a arithmetic cycle. Procedure `do_grep()` passes `&defstab->stbp_val` into `savesptr()`. Because of part 1, this could be `str_undef + 4`. Procedure `savesptr()` takes its incoming parameter and assigns it into `*str`. Because of part 2, `*str` aliases to `str_undef`. Thus, cumulatively, `str_undef` may obtain `str_undef + 4`. Should `str_undef` point to an object it will point to all offsets of that object. The machine-level model results in the derivation of additional data flow. Section 5.5.2 will discuss the implications of this extra data flow. While undesired (given omiscient knowledge), this data flow is required in faithfulness to the machine-level model.

```
STR str_undef;
STAB *defstab;

 /* PART ONE */
STR *hfetch() { return &str_undef; }
STAB *stabent() {
    STAB *stab = (STAB*)hfetch();
    if (stab == (STAB*)&str_undef)
        return Nullstab;
    return stab;
}
int main() { defstab = stabent(); }

 /* PART TWO */
void str_free(STR *str) {
    if (str == &str_undef)
       return;
    froot = str;
}
STR *str_new() { return froot; }

 /* PART THREE */
void savesptr(STR **sptr) {
   str = str_new();
   *str = sptr;
}
int do_grep() {
   savesptr(&((*defstab)->stbp_val));
}
```

Figure 5.11 Source code extracted from the benchmark *134.perl* that results in a false, arithmetic cycle.

5.5 Empirical Results

The first goal of the evaluation is to show that the proposed approach to safe field-sensitivity described in Section 5.2 can be made efficient using the techniques from Section 5.4. To this end, we implemented three versions of the safe field-sensitivity: Enumerative (*ENUM*) directly implements the derivation rules from Section 5.2.2; Object-only (*OBJ*) applies strides to only objects detailed early in Section 5.4.2; and Full (*FULL*) applies coarse-grain strides to objects while fine-grain strides are applied to constraints, as proposed at the end of Section 5.4.2.

The second goal is to evaluate the efficiency and accuracy of safe field-sensitivity, specifically FULL. FULL will be compared against a field insensitive (*FI)* implementation that is inherently safe but not very accurate and a more traditional field sensitive (*UNS*) implementation that is not safe, but more accurate ("resolves smaller points-to sets") than FI. UNS models the effects of all field accesses, including implicit pointer arithmetic, but ignores explicit pointer arithmetic and array indexing. The results will show that FULL is much more accurate and *faster* than FI. They will also show that FULL is almost as fast as UNS though it does, in some cases, lose ground on accuracy ("resolves larger points-to sets").

5.5.1 Effect on efficiency

The following compares the efficiency of FI and UNS to the ENUM, OBJ, and FULL. It also includes data on the arithmetic cycle detection and the effect of online monitoring for detecting cycles.

ENUMERATION MODE. Table 5.1 lists the analysis time for four benchmarks using FI, UNS, ENUM, OBJ, and FULL. Complete enumeration of the constraints drastically increases the analysis time, sometimes by a factor of 1000. The additional cost is solely due to the increase in the number of constraints. This mode exhibits a lack of scalability for even these modestly sized benchmarks and motivates the use of a compressed representation, such as OBJ or FULL.

Table 5.1 Analysis time (seconds) comparing a field insensitive and a typical field sensitive algorithm against the proposed safe field sensitivity implemented using full enumeration, strides on objects, and strides on constraints and objects.

Benchmark	FI	UNS	Safe FS		
			ENUM	OBJ	FULL
008.espresso	133	8	>1800	81	8
132.ijpeg	135	0.5	13	875	0.6
197.parser	0.3	0.3	606	0.3	0.2
mpeg4dec	5	1.5	>1800	10	2

Object-Only Mode. The column OBJ shows the analysis time when strides are applied only to objects. The benchmarks *008.espresso*, *197.parser*, and *mpeg4dec* show substantially improved analysis times over ENUM. *132.ijpeg* exhibits an interesting reaction to the change. While previously performing well for ENUM, its analysis time increased dramatically when strides were applied to objects.

The degradation in *132.ijpeg*'s analysis time is explained by Table 5.2, which shows the average number of static loads that appear to access named and heap objects (for more discussion of this metric, see Section 5.5.2). For *132.ijpeg*, there is roughly a 100-fold increase in the number of loads perceived to access each program object. Compared to ENUM, this is a substantial degradation in the accuracy. The degradation is caused by the OBJ's response to a pointer-sized strided access to a critical, large data structure. This fine-grain access is applied to the object as a whole, leading to a loss of field-sensitivity for that data structure and, subsequently, a substantial over approximation of the call graph.

Table 5.2 Effect of using ENUM and OBJ on accuracy. Columns *lds/obj* show the average number of static loads that appear to access a particular object. The average is shown separately for named and heap objects. OBJ results in substantial loss of accuracy for *132.ijpeg*.

Benchmark	ENUM (lds/obj)		OBJ (lds/obj)	
	Nmd	Heap	Nmd	Heap
008.espresso	1.4	27	1.4	131
132.ijpeg	1.7	6.9	109	1353

Full Mode. We have seen that neither object strides alone nor constraint enumeration are satisfactory solutions. FULL implements a "hybrid" approach presented in Section 5.4.2. In particular, coarse-grain strides are applied directly to the objects, while fine-grain accesses are applied to constraint edges through the use of a universal stride. Applying fine-grain strides to constraints helps prevent fine-grain accesses from degrading the accuracy of large data structures, as occurred with *132.ijpeg* for OBJ.

Figure 5.12 shows the fraction increase (decrease) in analysis time when comparing UNS and FULL to a FI baseline. The four benchmarks *008.espresso*, *130.li*, *132.ijpeg*, and *jpg2Kdec* show dramatic analysis speedups when either field sensitive algorithm is used. This is due to a bad interaction between FI and CS where CS replicates heap locations in an attempt to better isolate interactions while FI creates "leaks" that render the numerous heap locations no longer isolated. Essentially, CS causes the problem size to grow while FI corrupts the result. In fact, the benchmark *jpg2Kdec* failed to complete within 30 min for FI. To aid interpretation of the data, the field sensitive time for these four benchmarks has been included and is shown in brackets near the x-axis.

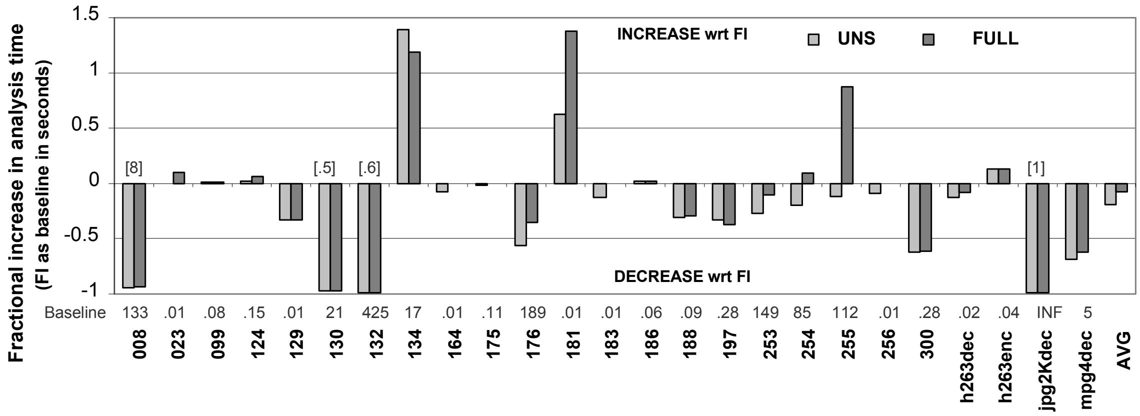


Figure 5.12 Fraction increase (decrease for negative) in time when comparing UNS and Safe FS FULL to a FI baseline. FI Baseline times are shown between the benchmark labels and the graph. Benchmarks *008.espresso*, *130.li*, *132.ijpeg*, and *jpg2Kdec* show dramatic analysis speedups when any field sensitive algorithm is used. To aid interpretation of the data, the approximate FS times for these four benchmarks are shown in brackets near the x-axis.

FULL shows the largest fractional increase in analysis time for the benchmarks *181.mcf* and *255.vortex*. The total time for *181.mcf* is around 1 ms, so this change is inconsequential. The increase for *255.vortex* is largely due to a loss of accuracy which will be detailed in Section 5.5.2. Generally speaking, Safe FS FULL results take no longer than traditional UNS results.

Cycle Detection. Figure 5.13 shows the fraction of variables found to be involved in local arithmetic cycles, the fraction of objects to which a stride was applied, and the fraction of points-to edges to which a universal stride was applied. On average, about 7% of all variables are involved in pointer-arithmetic cycles. This large percentage makes it clear why, from the perspective of safety, it is important to appropriately model cycles and to handle their consequences efficiently. The addresses of many objects become involved in these arithmetic cycles, resulting in the application of a coarse-grain stride

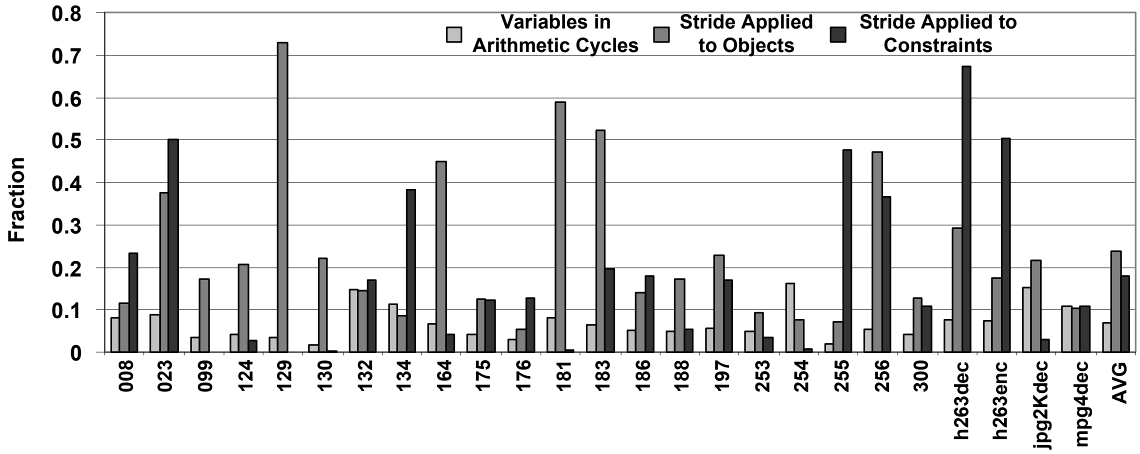


Figure 5.13 Fraction of variables involved in skew cycles, fraction of objects to which a stride was applied, and the fraction of points-to edges to which a stride was applied.

to 23% of the objects and a universal stride to 18% of all address coinstraints. A large fraction of program data flow interact with one of these cycles, further illuminating why explicit enumeration of strided accesses can explode.

Online Monitoring. Section 5.4.3 walked through an interprocedural arithmetic cycle in *134.perl* that exists for both UNS and FULL. Experiments on *134.perl* showed that this cycle leads to a significant increase in the number of constraints and the analysis process no longer completed within the alotted 30 min. Online monitoring detects the impending constraint explosion and reduces the analysis time to about 40s.

5.5.2 Effects on accuracy

This section presents the impact of Safe FS FULL on the quality of the resultant pointer information. For a measure of accuracy, we use the number of loads the analysis determines may access a particular object. An average is taken across all objects to

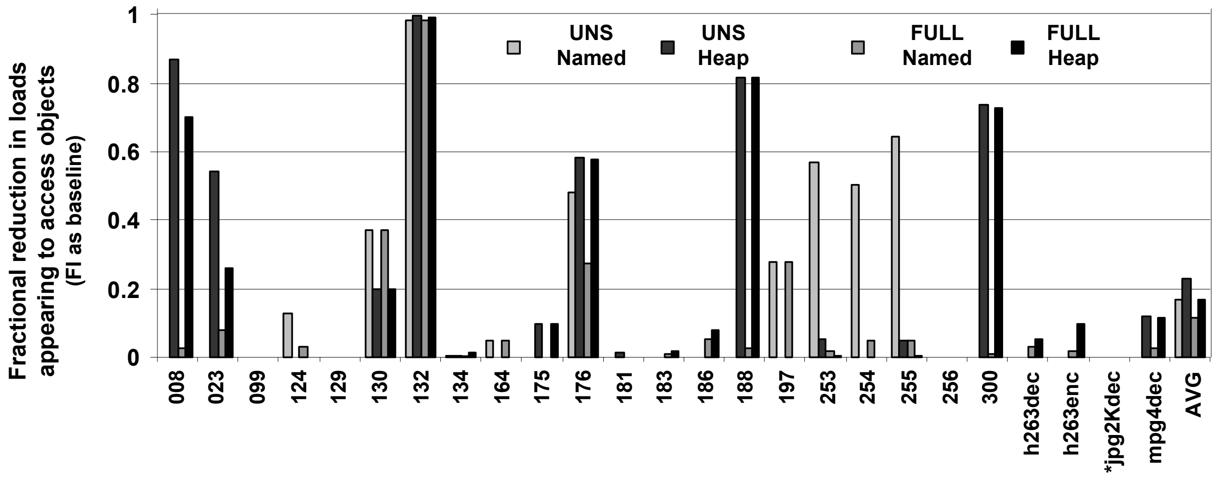


Figure 5.14 Fractional decrease in the number of loads perceived to access program objects when compared to the results from a FI baseline. The benchmark *jpg2Kdec* failed to complete for FI and thus its FS data has been excluded.

obtain an average loads-per-object. An analysis result is better if it reduces loads-per-object. We also compute a similar metric with respect to stores, but these have been omitted since the trends were similar.

Figure 5.14 shows the fractional decrease in loads-per-object for UNS and FULL when compared to the results from a FI baseline. Note that the benchmark jpg2Kdec failed to complete within the alotted 30 min for FI mode; thus, its FS data has been excluded. The impact of FULL on accuracy is mixed. In many instances, FULL has little effect on the results. However, the benchmarks *253.perl*, *254.gap*, and *255.vortex* show a significant loss of accuracy. The use of a global pool allocator for *253.perl* and *254.gap* causes a single abstract location to represent almost all heap allocated objects. Subsequent application of a mix of strides to this object effectively results in a loss of field sensitivity for most heap references. The heap locations allocated by *255.vortex* are immediately accessed by expressions similar to `++ChunkAddrPtr` and `:= *(++ChunkAddrPtr)` which leads to most

Table 5.3 Effect of Safe FS FULL on CI and CS results. FULL impacts the analysis time (a) of CI more that CS for *132.ijpeg* and *008.espresso* and degrades the accuracy (b) of CI more that CS for *132.ijpeg* and *130.li*.

Benchmark		UNS	Safe FS FULL
008.espresso	CI	0.2	12
	CS	8	207
132.ijpeg	CI	0.7	13
	CS	0.5	0.6

(a)

Benchmark		UNS lds/obj		Safe FS FULL lds/obj	
		Named	Heap	Named	Heap
132.ijpeg	CI	6	1087	42	2205
	CS	1.7	6.9	1.7	11
130.li	CI	60	557	76	712
	CS	50	395	50	395

(b)

heap accesses obtaining a universal stride, as can be seen for *255.vortex* in Figure 5.13. Again, this leads to a loss of field sensitivity for heap references.

5.5.3 Interaction between FULL and CI, CS

Table 5.3 compares the effect of Safe FS FULL mode on CI and CS results. Table 5.3(a) compares analysis time and Table 5.3(b) accuracy. In these instances, the impact of FULL mode on a CS algorithm is less than its impact on a CI one. This result is not unexpected, since a CS algorithm generates fewer false relationships and thus fewer strided accesses to objects. Since a context sensitive algorithm can better isolate the data flow between objects, it can also be more resilient to a more conservative model of field sensitivity.

6. COMPREHENSIVE EMPIRICAL STUDIES

6.1 Efficiency

Figure 6.1 compares the amounts of time necessary to perform four different kinds of pointer analysis. At a minimum, all analyses use a per-call-site heap model. The blue line (diamond marker) plots the analysis time necessary to perform a simple context- and field-insensitive pointer analysis. For a few benchmarks, such as *130.li* and *132.ijpeg* this is the slowest analysis. These benchmarks are a good example of how the cost of extra derivations can result in a simple, inaccurate analysis being more expensive than an accurate one.

The pink line (square marker) represents a field-sensitive analysis and the green line (triangle marker) a field- and context-sensitive analysis. Both of these analyses are very fast. For these two pointer analyses, the benchmarks *176.gcc*, *253.perlbmk*, and *254.gap* stand out as the most costly benchmarks requiring about 25, 100, and 70 s, respectively, for the field- and context-sensitive analysis. The analysis time for the rest of the benchmarks range from 0.01 to about 5 s.

The orange line (cross marker) plots the analysis time when exhaustive heap specialization is performed. This has the biggest impact on *008.espresso*, *134.perl*, *176.gcc*, and *255.vortex*. All of these benchmarks show a large increase in the number of modeled heap objects and, therefore, a corresponding increase in analysis time. *008.espresso* and

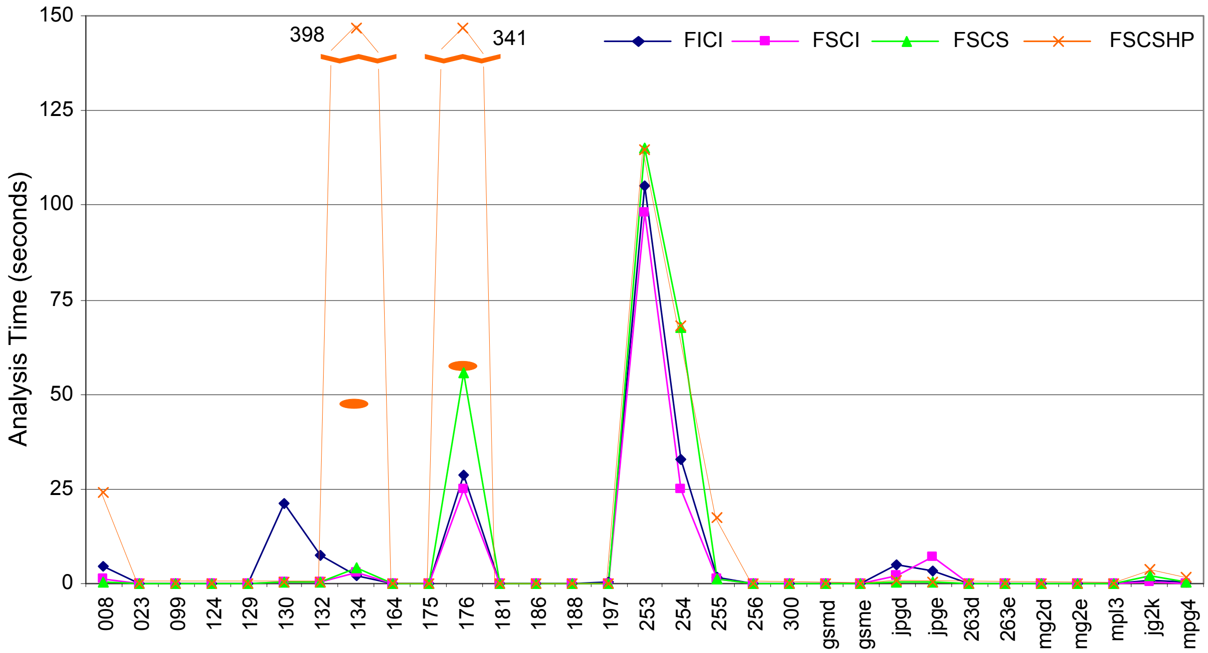


Figure 6.1 Time necessary for Fulcrato perform four different analyses: field-insensitive, field-sensitive, field- and context-sensitive, and field-, context-, and heap-sensitive.

255.vortex require roughly 25 and 20 s, respectively. *134.perl* and *176.gcc* jump to almost 400 s. Depending on the importance of exhaustive heap specialization, Fulcra supports the placing a hard limit on the amount of specialization performed. If a limit of two heap generations is used, the analysis time necessary for *134.perl* and *176.gcc* drops to 50 and 60 s, respectively, and is shown by the orange ovals in the figure.

To provide better resolution on the cost of different pointer analysis configurations, Figures 6.2 and 6.3 show relative increases in analysis time. Figure 6.2 compares a field-sensitive analysis to a field-insensitive analysis. Overall, by better modeling actual data flow, field-sensitivity tends to decrease analysis time.

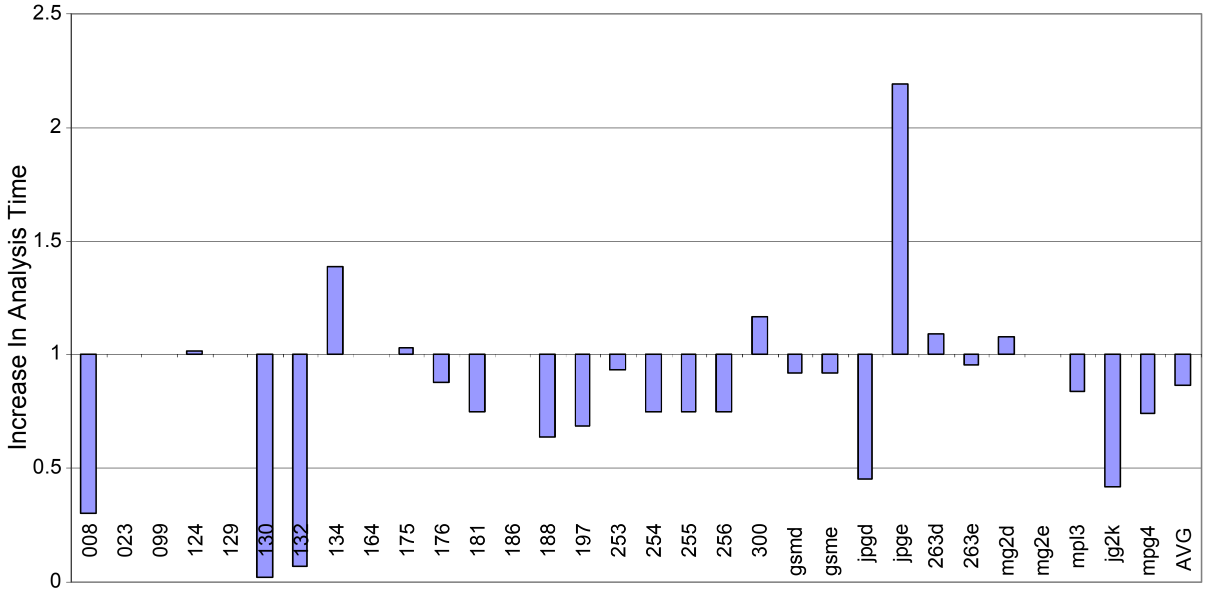
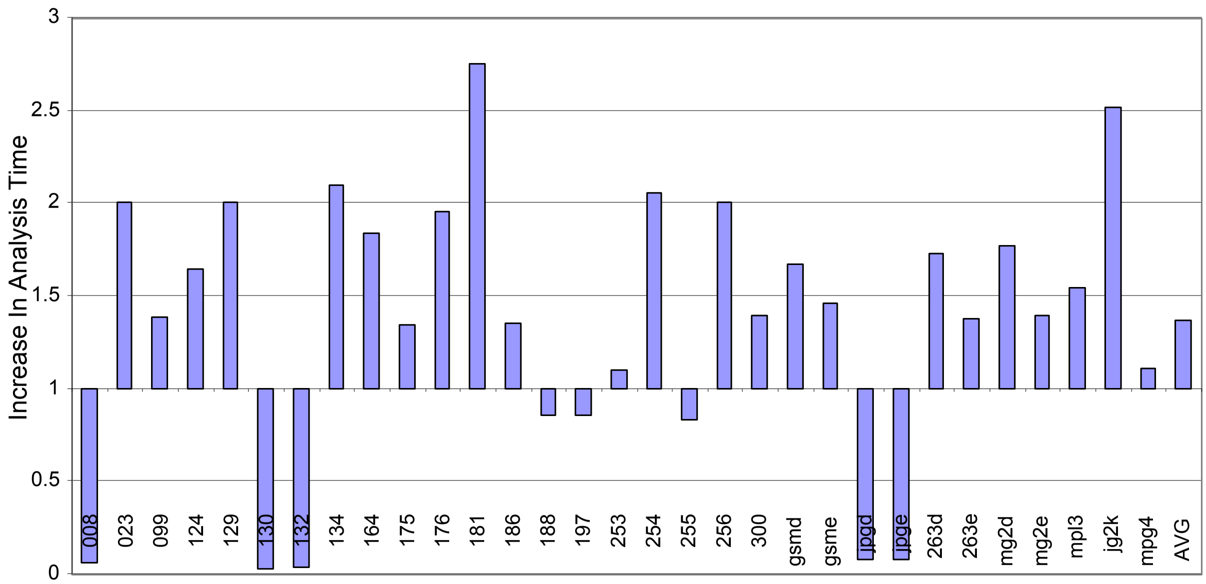


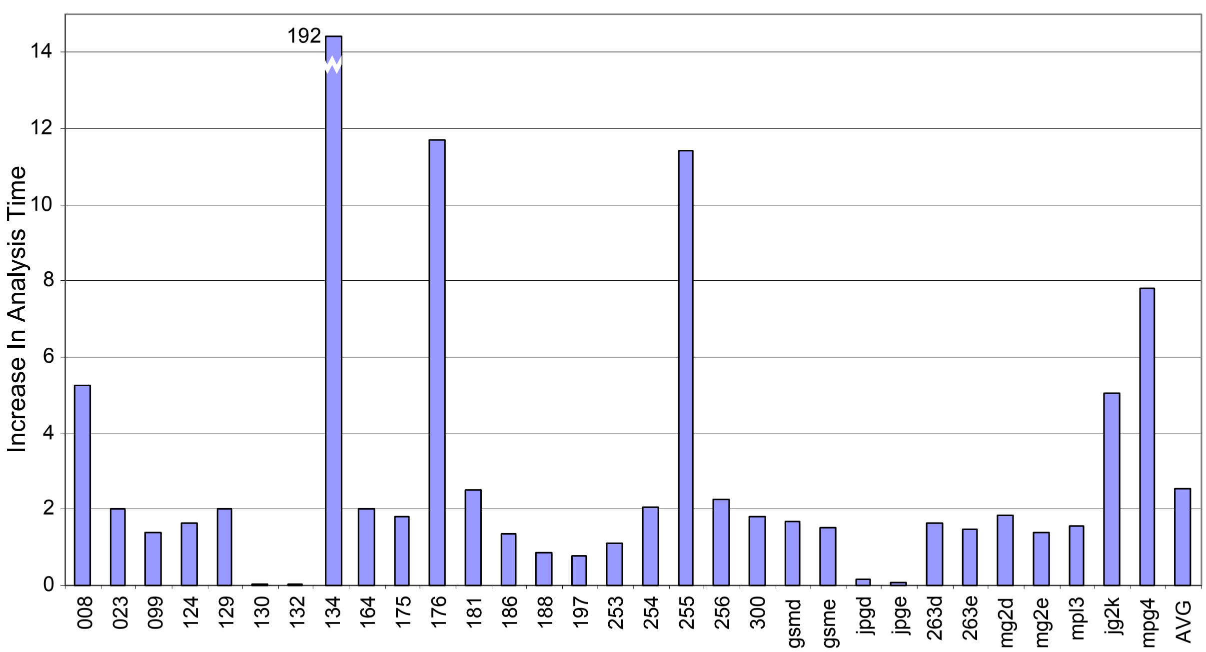
Figure 6.2 Factor increase in analysis time for a field-sensitive, context-insensitive analysis over a field-insensitive analysis.

Figure 6.3(a) and (b) plot the relative increase in analysis time for context-sensitive and heap-sensitive analyses against a field-sensitive but context-insensitive analysis. On average, context-sensitivity adds about 40% to the analysis time, but can triple analysis time in some instances. Note, however, for *jpg2Kdec* and *181.mcf* the baseline analysis times are less than or near 1 s. A large fraction of the increase is likely due to differing baseline overheads between the two analyses.

Heap specialization shows a more significant cost, about 2x on average if *134.perl* is excluded. However, it is clear that *134.perl*, *176.gcc*, and *255.vortex* are impacted the most with *134.perl* standing out. The issue with *134.perl* is that heap specialization creates many heap locations that, in the end, differ little from each other. Unfortunately, the heap recombination is unable to resolve the problem before expending a lot of analysis time.



(a)



(b)

Figure 6.3 Factor increase in analysis time for (a) a field-sensitive, context-sensitive and (b) a field-sensitive, context-sensitive, and heap sensitive analysis against a baseline field-sensitive, context-insensitive analysis.

Benchmark	Fulcra	Cheng	Foster
008.espresso	24	9	995
023.eqntott	.02	2	12
129.compress	.01	1	2
130.li	.50	1332	9981
176.gcc	50-350	hours	-
253.perlbmk	120	months	-

Table 6.1 Pointer analysis times for benchmarks common to multiple publications.

To provide a frame of reference for Fulcra's analysis times, Table 6.1 presents data for benchmarks common to Cheng and Hwu [19] and Foster et al. [20] publications. With the exception of Cheng's analysis of *008.espresso*, Fulcra is orders of magnitude faster than both. The extra analysis time required for *008.espresso* is likely due to Fulcra's more complicated modeling of field-sensitivity and a safer modeling of heap objects. Cheng's implementation makes extensive use of types to filter out unrealizable data flow but can lead to incorrect analysis results. Despite this, the analysis time swells substantially for *176.gcc* and *253.perlbmk* while remaining under control for Fulcra.

6.2 Accuracy

Figure 6.4 compares the accesses-per-object obtained by five different analysis configurations. As in the timing runs, all configurations minimally model per-allocation site heap specialization *h0*. The letter *f* means field sensitive, *c* context sensitive, and *h* exhaustively heap sensitive. The dark bars mark the most accurate configuration and also provide visual separation between benchmarks. Because of the large range of results for

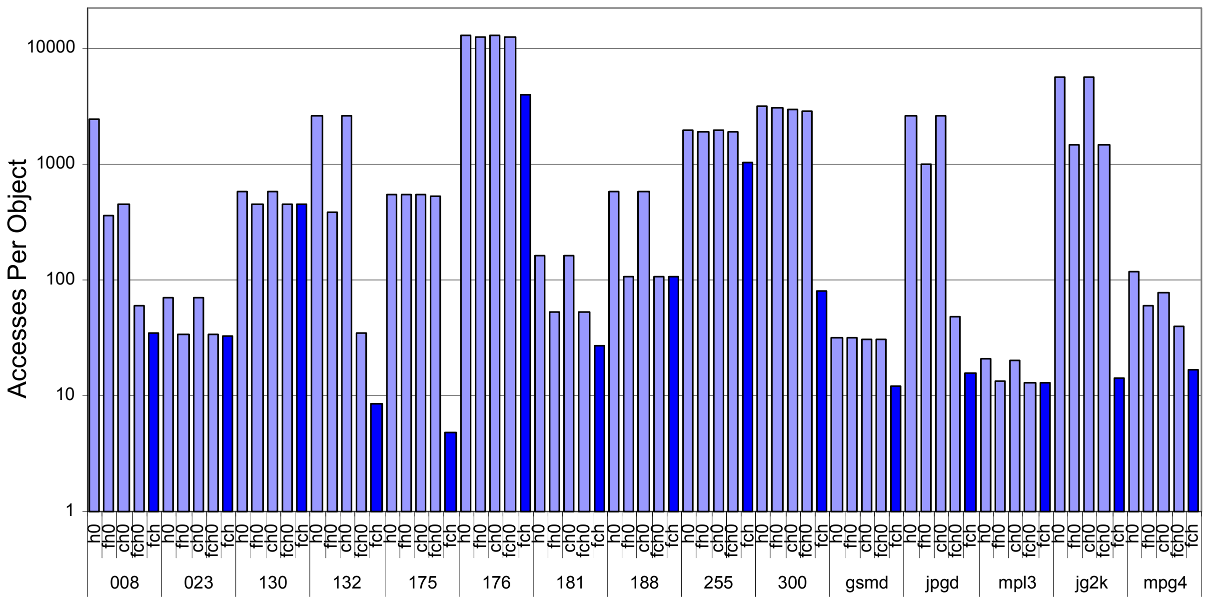


Figure 6.4 Effect of different analysis configurations on the number of access-per-object observed.

some benchmarks, a logarithmic scale is used. There are two important points. First, in many cases the analysis configuration can cause an order of magnitude, if not orders of magnitude, difference in the result accuracy. Second, in some cases there is little difference until the most accurate form of the analysis is performed. By themselves, field-, context-, or heap- sensitivity are not individually sufficient but, instead, a convergence of the three is necessary to realize the gains.

Figures 6.5 and 6.6 provide additional resolution into the impact of different analysis configurations on accuracy. Each plots the fractional decrease in accesses-per-object observed and separates the statistic for named and heap objects.

Figure 6.5 shows the decrease in accesses-per-object for a field-sensitive algorithm over a field-insensitive one. On average, field-sensitivity results in a 20% reduction in accesses-per-object. However, the actual results tend to be polar, with a few benchmarks

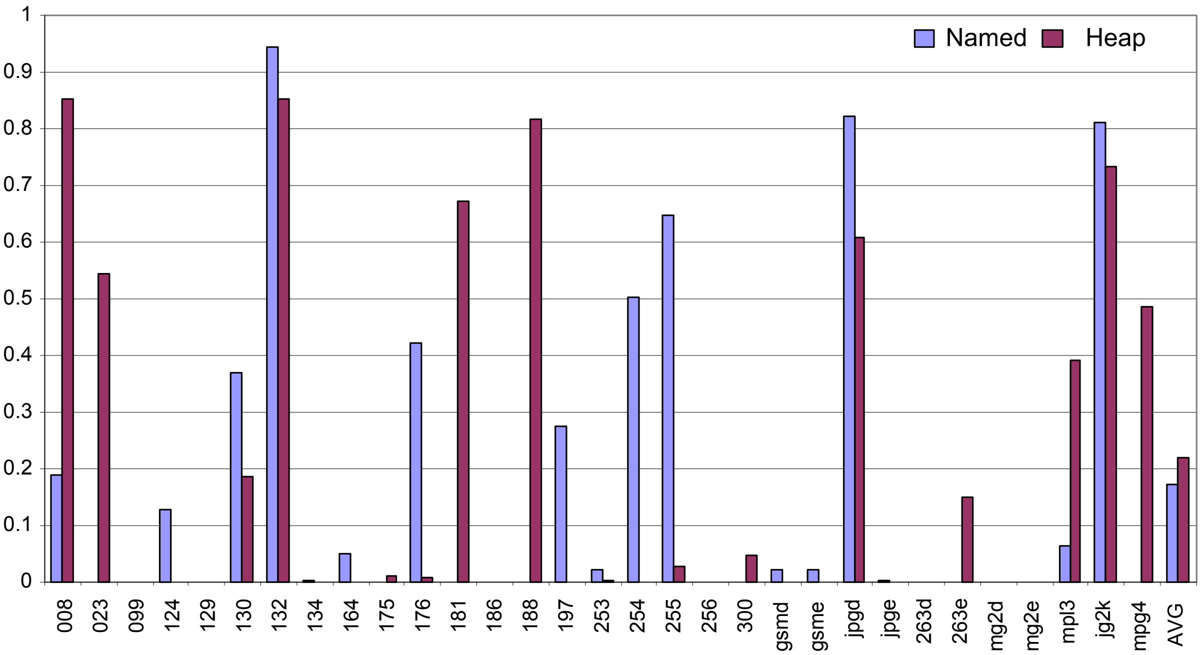
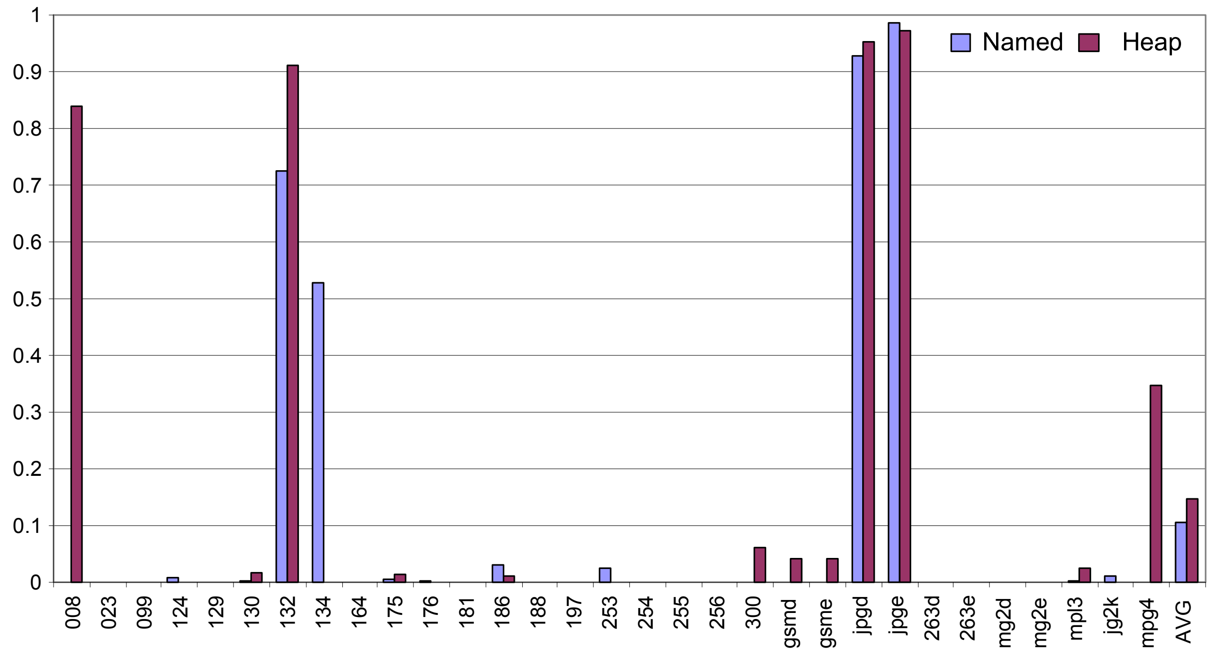
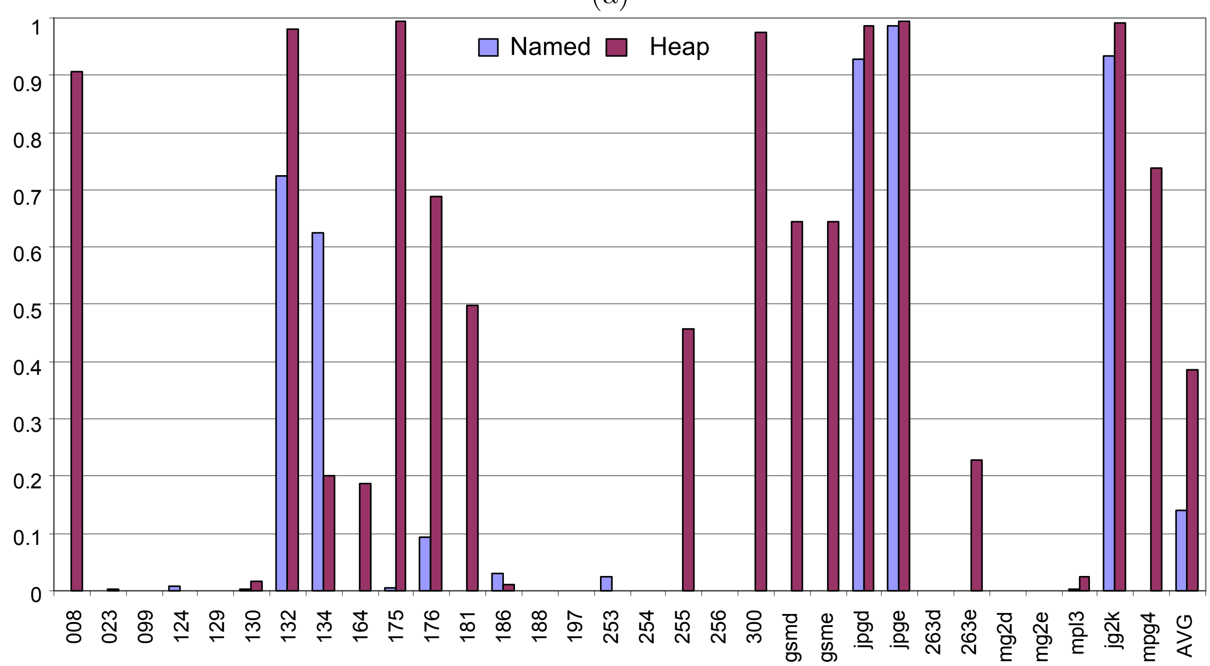


Figure 6.5 Decrease in access-per-object observed for a field-sensitive, context-insensitive analysis over a field-insensitive analysis.

benefiting little and others showing substantial reductions. Figure 6.6(a) shows the fraction decrease for a context-sensitive algorithm against a context-insensitive one (both are field-sensitive). Context-sensitivity alone shows large improvements for only six benchmarks. However, when heap specialization is performed the results improve dramatically, as shown in Figure 6.6(b), with roughly half of the benchmarks showing substantial reductions in accesses-per-object. The average decrease goes from 10% named and 15% heap without heap sensitivity to 15% named and 40% heap with it.



(a)



(b)

Figure 6.6 Decrease in access-per-object observed for (a) a field-sensitive, context-sensitive and (b) a field-sensitive, context-sensitive, and heap sensitive analysis against a baseline field-sensitive, context-insensitive analysis.

7. FUTURE DEVELOPMENT

7.1 Summary Refinement

The summaries formed by Fulcra are complete, context independent summaries. This means that summarization can be performed once for a particular procedure and universally used. However, it is possible that the set of contexts encountered throughout the entire program is smaller than the set of all possible contexts. If such conditions are common, the solution preview capability can be leveraged to determine the set of known contexts. This context information could be used in place of the holding property to mark the parts of a procedure that contain external references and, hopefully, lead to the creation of a smaller summary.

These partial summaries are still context independent. They are not for a particular call path context but are formed for all *observed* call path contexts instead of all *possible* call path contexts. The cost is that, should the bottom-up process result in a change to the input contexts, new summaries may need to be generated. The philosophy behind this idea is similar to *counter-example directed refinement*, a popular scheme in model checking [37].

```
red(p) {
    heap1 := &heap2;
    s := &heap1;
    *p := s;
    y := &heap3;
    *p := y;
}
```

(a)

```
red(p) {
    heap1 := &heap2;
    s := &heap1;
    *p := s;
}
```

(b)

Figure 7.1 Code example (a) for which more aggressive redundancy removal could obtain a smaller summary (b) by leveraging the fact that heap location `heap1` is a superset of `heap3`.

7.2 Redundancy Removal

One theme throughout this dissertation is the importance and benefits of keeping summaries small. The redundancy removal algorithm used is essentially that proposed in [23]. While good, the algorithm is not as aggressive as it could be.

Consider the example in Figure 7.1(a). The location `&heap1` is a provable superset of the location `&heap3`; thus, the summary could be reduced to the code in Figure 7.1(b). However, since `&heap1` and `&heap3` are not identical the standard redundancy removal is not capable of this transformation.

There are also opportunities to expose additional redundancy in the presence of aggregate objects using a process termed *fragmentation*. Consider the code in Figure 7.2(a). With the structure `st` left intact, there is no opportunity for redundancy removal. However, if the structure can be fragmented into two pieces, shown in Figure 7.2(b), lines 1 and 3 become redundant with lines 4 and 5.

```
{
    ...st.f1 := a;
    st.f2 := b;
    c := st.f1;
    x := a;
    c :=.x}
```

(b)

```
{
    ...tmp1 := a;
    tmp2 := b;
    c := tmp1;
    x := a;
    c :=.x}
```

(b)

Figure 7.2 Code example (a) for which structure fragmentation could expose additional redundancy by transforming the code to that shown in (b).

7.3 Hybrid Inclusion-Unification

Fulcra already contains select applications of a unification style analysis process. For example, redundancy removal is performed both on procedure summaries and, prior to analysis, the procedures themselves. Redundancy removal performs unification when it is provable that such unification will not change the result. Furthermore, heap recombination presented in Section 4.7 performs unification on heap objects when they are statistically very similar. There is opportunity for a more general and statistical approach to unification. This could improve the scalability of the analysis while leaving it less fragile to a cascade of unification steps.

7.4 Context Sensitive Recursion

In most frameworks, recursion is handled in a context insensitive manner. Recall that, before summarizing a procedure, the summaries for its callees are specialized and inlined. In the presence of a call graph cycle, complete callee summaries cannot be guaranteed available. This means that an algorithm would have to iterate around the

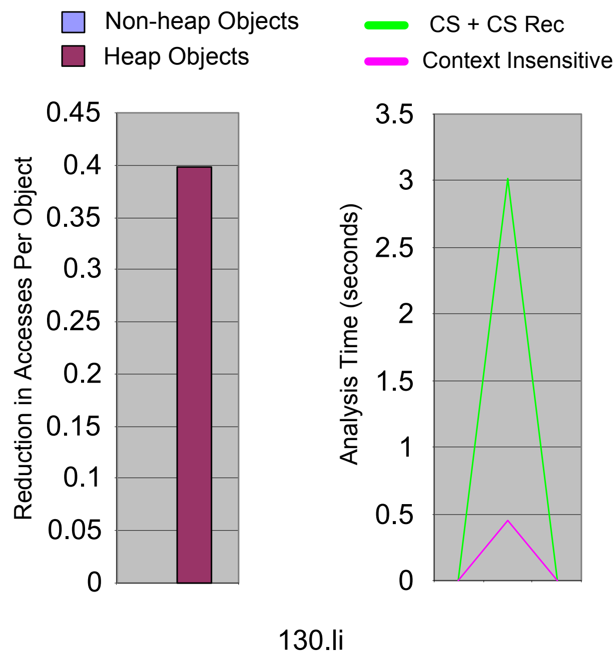


Figure 7.3 Effect of context-sensitive recursion on analysis time and accuracy of *130.li*.

cycle, continually expanding summaries until a steady state is reached. Not only could summaries become very large, but, in an Andersen's style formulation or if heap cloning is performed, this process may diverge and never complete.

Fulcra's ability to selectively turn off context sensitivity could be used as a throttle to keep summaries small during attempts at recursive context sensitivity. In the worst case, the cycle would end up being treated context insensitively. In the best case, no throttling would be necessary. Using a rudimentary implementation, Figure 7.3 shows that recursive context sensitivity benefits the analysis of program *130.li*.

```
1:  switch(format) {
2:     case BMP: dest = jinit_bmp(&cinfo); break;
3:     case GIF: dest = jinit_gif(&cinfo); break;
7:  }
8:  (*dest->start_output)(&cinfo, dest);
```

Figure 7.4 Excerpt from *jpegdec* where an analysis result can benefit from the correlation between the indirect call at line 8 and its parameter, both involving variable `dest`.

7.5 Control Correlation

While motivating the use of an Andersen's style formulation, Section 2.3 alluded to the possibility of improving upon the analysis results for code from *jpegdec*, previously shown in Figure 2.3. An excerpt from this example is shown here in Figure 7.4.

At line 8, an indirect function call is controlled by the object pointed-to by `dest`. Furthermore, the second parameter to this call is also `dest`. Even though the framework is flow-insensitive, it is possible to leverage the fact that, while `dest` may contain a number of references, it can contain only one at any given moment. For example, assume that `dest` may point to object `A` with a `start_output` field of `callA()` and object `B` with a `start_output` field of `callB()`. By leveraging the known correlation between the indirect call site and the second parameter, procedure `callA()` can only obtain `A` as its input parameter and `callB()` can only obtain `B`.

Such code sequences often occur in programs written to be configurable. For the program *jpegdec*, leveraging this correlation can result in substantial improvements in accuracy.

7.6 Partial Programs

The algorithms presented assume the existence of an entire program. In a non-experimental environment, programs are often composed of separately compiled libraries or modules. Pointer analysis will often have to be performed on program fragments with missing callers, callees, or both. The lack of callers or callees makes it important to differentiate the pointer information that is complete from that which is incomplete.

If a caller is missing, this can be done by applying a special, artificial input to each unknown parameter or accessible global variable. From a modeling perspective this would be an address constraint from a node that points to itself. During constraint solution process, the reach of this input would be resolved as well.

Missing callees can be handled in a similar fashion. A special constraint can be applied to the actual parameters at the missing callee's call site and to accessible global variables. The incompleteness of global variables would flow context insensitively due to the global qualifier and the potential effect on a callee could be recorded concisely in the procedural summary. Note that, from a summarization perspective, it is enough to know that some unknown effect exists. It is not important to try and model the particulars of that effect; thus, there may be additional compaction opportunities during the backtracing and redundancy removal step.

8. CONCLUSION

This dissertation detailed a qualified constraint based approach to context-sensitivity that provides the backbone for an efficient and flexible pointer analysis framework. Furthermore, it detailed how to leverage the qualified constraint framework to obtain heap sensitivity as well as five mechanisms that Fulcra employs to contain the costs of heap sensitivity. A flexible form of field sensitivity capable of safely modeling type-unsafe C was presented along with the algorithmic modifications necessary to obtain a jointly context- and field-sensitive framework. Finally, a comprehensive evaluation demonstrated that, for the applications used for evaluation, the algorithms provided all analysis results in under eight minutes with most requiring a second or less. Using the most aggressive analysis configuration, the accuracy of the results for many applications showed orders of magnitude improvement over previous work.

REFERENCES

[1] M. Hind and A. Pioli, "Which pointer analysis should I use?," in *Proceedings of International Symposium on Software Testing and Analysis*, 2000, pp. 113–123.

[2] M. Hind, "Pointer analysis: Haven't we solved this problem yet?," in *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2001, pp. 54–61.

[3] L. O. Andersen, "Program analysis and specialization for the C programming language,", Ph.D. dissertation, Datalogisk Institut, University of Copenhagen, 1994.

[4] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, 1996, pp. 32–41.

[5] A. Milanova, A. Rountev, and B. Ryder, "Parameterized object sensitivity for points-to and side-effect analyses for Java," in *Proceedings of International Symposium on Software Testing and Analysis*, 2002, pp. 1–11.

[6] R. P. Wilson and M. S. Lam, "Efficient context-sensitive pointer analysis for C programs," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1995, pp. 1–12.

[7] R. Chatterjee, B. G. Ryder, and W. A. Landi, "Relevant context inference," in *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, 1999, pp. 133–146.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadek, "An efficient method of computing static single assignment form," in *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, 1989, pp. 25–35.

[9] R. Hasti and S. Horwitz, "Using static single assignment form to improve flow-insensitive pointer analysis," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998, pp. 97–105.

[10] M. Das, B. Liblit, M. Fähndrich, and J. Rehof, "Estimating the impact of scalable pointer analysis on optimization," in *Proceedings of the Static Analysis Symposium*, 2001, pp. 260–278.

[11] D. Liang and M. J. Harrold, "Efficient computation of parameterized pointer information for interprocedural analyses," in *Proceedings of the Static Analysis Symposium*, 2001, pp. 279–298.

[12] M. Das, "Unification-based pointer analysis with directional assignments," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000, pp. 35–46.

[13] C. Lattner and V. Adve, "Data structure analysis: A fast and scalable context-sensitive heap analysis," University of Illinois at Urbana Champaign, Tech. Rep. UIUCDCS-R-2003-2340, 2003.

[14] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, 1995, pp. 49–61.

[15] M. Emami, R. Ghiya, and L. J. Hendren, "Context-sensitive interprocedural points-to analysis in the presence of function pointers," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1994, pp. 242–256.

[16] M. Hind, M. Burke, P. Carini, and J.-D. Choi, "Interprocedural pointer alias analysis," *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 4, pp. 848–894, 1999.

[17] W. Landi and B. G. Ryder, "A safe approximation algorithm for interprocedural pointer aliasing," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1992, pp. 235–248.

[18] E. Ruf, "Context-insensitive alias analysis reconsidered," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1995, pp. 13–22.

[19] B.-C. Cheng and W.-M. Hwu, "Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000, pp. 57–69.

[20] J. S. Foster, M. Fähndrich, and A. Aiken, "Polymorphic versus monomorphic flow-insensitive points-to analysis for C," in *Proceedings of the Static Analysis Symposium*, 2000, pp. 175–198.

[21] D. Liang and M. J. Harrold, "Efficient points-to analysis for whole-program analysis," in *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1999, pp. 199–215.

[22] E. M. Nystrom, H.-S. Kim, and W.-M. Hwu, "Bottom-up and top-down context-sensitive summary-based pointer analysis," in *Proceedings of the Static Analysis Symposium*, 2004, pp. 112–138.

[23] H.-S. Kim, "Context-sensitive pointer analysis based on procedural summaries,", Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2004.

[24] J. Rehof and M. Fähndrich, "Type-based flow analysis: from polymorphic subtyping to CFL-reachability," in *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, 2001, pp. 54–66.

[25] E. M. Nystrom, H.-S. Kim, and W.-M. Hwu, "Importance of heap specialization in pointer analysis," in *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2004, pp. 43–48.

[26] A. Rountev, A. Milanova, and B. Ryder, "Points-to analysis of Java using annotated constraints," in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2001, pp. 43–55.

[27] D. J. Pearce, P. H. J. Kelly, and C. Hankin, "Efficient field-sensitive pointer analysis for C," in *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2004, pp. 37–42.

[28] B. Steensgaard, "Points-to analysis by type inference of programs with structures and unions," in *Proceedings of the ACM SIGPLAN 82 Symp. on Compiler Construction*, 1996, pp. 136–150.

[29] S. H. Yong, S. Horwitz, and T. Reps, "Pointer analysis for programs with structures and casting," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1999, pp. 91–103.

[30] R. Ghiya, D. Lavery, and D. Sehr, "On the importance of points-to analysis and other memory disambiguation methods for C programs," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2001, pp. 47–58.

[31] S. Debray, R. Muth, and M. Weippert, "Alias analysis of executable code," in *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, 1998, pp. 12–24.

[32] Q. Lhoták and L. J. Hendren, "Scaling java points-to analysis using SPARK," in *Proceedings of the ACM SIGPLAN 82 Symp. on Compiler Construction*, 2003, pp. 153–169.

[33] J. Whaley and M. Rinard, "Compositional pointer and escape analysis for Java programs," in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1999, pp. 187–206.

[34] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer, "CCured in the real world," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003, pp. 232–244.

[35] G. C. Necula, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy code," in *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, 2002, pp. 128–139.

[36] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken, "Partial online cycle elimination in inclusion constraint graphs," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1998, pp. 85–96.

[37] E. Clarke, O. Grunmberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *Journal of the ACM*, vol. 50, no. 5, pp. 752–794, 1997.

AUTHOR'S BIOGRAPHY

Erik Nystrom was born in Colorado Springs, Colorado. He was the fifth of six children to Perry Everett Nystrom and Elizabeth Ann Nystrom. He grew up in Raleigh, N.C., where he attended North Carolina State University. During his undergraduate studies he worked for both Nortel Networks and IBM and was the holder of both the Lockheed-Martin and Barry M. Goldwater National Scholarships. He graduated Summa cum Laude from North Carolina State University with a B.S. in Computer Engineering and was the first recipient of NCSU's Outstanding Computer Engineer Award. For his graduate studies, Erik attended the University of Illinois at Urbana-Champaign, where he received an M.S. and a Ph.D in Electrical Engineering in 2002 and 2005, respectively. During his graduate studies, he worked for Intel and was the holder of both the National Science Foundation and Intel Doctoral Fellowships.