Static Dependence Analysis in an Infrastructure for Automatic Parallelization

Nick P. Johnson

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science

Adviser: Professor David I. August

September 2015

Abstract

Now that parallel architectures are common, software must exploit multiple cores to fully utilize hardware resources and achieve efficient execution. Restructuring applications for explicit parallelism requires developers to reason about low-level details to avoid concurrency bugs and achieve parallel performance. Automatic thread extraction relieves developers of this arduous task.

This dissertation presents a compiler *middle-ware* for automatic thread extraction—analyses and transformations that allow the compiler to deliver parallel performance for sequentially-specified input programs. This middle-ware reconsiders the compilation infrastructure to present alternative technologies better suited to the needs of automatic thread extraction. Specifically,

Collaborative Dependence Analysis: Since no analysis algorithm can precisely decide all analysis facts, this dissertation combines simple analysis algorithms into a *collaborative ensemble* algorithm: the ensemble algorithm can disprove dependence queries which no member disproves alone. Collaboration enables *factored* development, which prescribes the development of small, orthogonal analysis algorithms tailored to the needs of analysis clients. Results demonstrate independently-developed analysis algorithms collaborating to solve complex, multiple-logic queries. Further, results demonstrate a large impact on performance: for some benchmarks, analysis strength is the difference between $11\times$ slowdown and $28\times$ speedup on 50 cores.

Scaling Analysis to Larger Scopes: The infrastructure builds around Parallel-Stage Decoupled Software Pipelining (PS-DSWP) thread extraction. PS-DSWP targets large, hot program scopes to overcome the non-recurring overheads of parallel execution. However, as scopes grow, the burden of analysis becomes prohibitive. This dissertation contributes a faster algorithm to compute a dependence graph to drive PS-DSWP. This algorithm identifies dependence edges which *cannot* affect PS-DSWP. It skips dependence analysis queries pertaining to unimportant edges, reducing analysis time—or allowing more expensive anal-

ysis algorithms—for the remaining, important queries. Evaluation demonstrates that the algorithm computes the $\mathrm{DAG}_{\mathrm{SCC}}$ twice as fast using half as many dependence analysis queries without sacrificing analysis precision.

Incorporating Speculation into the Compilation Pipeline: A parallelization system may speculate various properties to enable thread extraction. This dissertation presents design patterns to simplify development of novel speculation types.

The dissertation integrates these contributions into a robust and flexible middle-ware upon which many works are built.

Acknowledgments

I thank my wife, Daya, for her patience, understanding and support through this all. Thank you, love, for standing with me. I thank our baby boy, Murland Harris Johnson, for reminding me of the beauty of life.

I thank my adviser, Professor David I. August, for his support over the years. He has taught me the processes of research and publishing, and has approached every problem with an *it can be done* attitude. I thank the the faculty on my dissertation committee: Professors Andrew Appel, David Walker, David Wentzlaff, and Jae W. Lee (SKKU). I would like to give a second thanks to Professors Wentzlaff and Lee for serving as readers on my committee.

Additionally, I thank the entire Liberty Research Group for the innumerable ways they have helped me over the years. Beyond collaboration, they have made graduate school fun. I thank Jialu Huang, Thomas Jablin, Hanjun Kim, Prakash Prabhu, Arun Raman, and Yun Zhang who greeted me at Princeton and welcomed me into Liberty Research. I thank Stephen Beard, Jordan Fix, Deep Ghosh, Feng Liu, Taewook Oh, and Matt Zoufaly, whom I greeted in turn. I thank Kevin Fan, Jae Lee, and Ayal Zaks. I thank Scott Mahlke for collaborating on our CGO'12 paper.

I thank those who have on several occasions helped me by reading preliminary drafts of my paper submissions, listening to practice talks, or discussing a research idea as it develops, including Andrew Appel, Gordon Stewart, Lennart Beringer, Jude Nelson, C.J. Bell, Sid Sen, Sushant Sachdeva, and Chris Monsanto. I extend additional thanks to Stephen Beard and Jordan Fix for commenting on drafts of this dissertation.

I thank the many friends I was lucky to meet at Princeton: Aleksey Boyko, Dan Reynolds, Katy Ghantous, Hjalmar Turesson, Olga Rodriguez Sierra, Kosmos Houdini Turesson Rodriguez, Darshana Narayanan, Waiyee Chiong, Jeff Ames, Sara Vantournhout, Wouter Rock, Leah Owens and Zach Smith. I thank the Cyclab: Sean Gleason, Emily Sullivan, Don Snook, Colin McDonough, and David Hocker. I thank Ken Steiglitz for sharing

walks around Princeton's campus and not talking about research. I thank the great community that is Butler Apartments; incredibly, the least insulated structures host the warmest neighborhood in all of Princeton.

I thank Princeton University for creating such an amazing environment for research, and for the many ways Princeton supports its graduate students. Dissertation bootcamp was very helpful in composing this document. Princeton's Terascale Infrastructure for Groundbreaking Research in Engineering and Science (TIGRESS) provided large clusters on which this work was evaluated.

I thank the Siebel Scholars program for their recognition and generous support during my fifth year of graduate school. I acknowledge the generous funding that has supported this work, including: "AAAC Architecture Aware Compiler Environment" (BAE Systems/AF award #077956 (PRIME FA8650-09-C-7918), Award dates: 5/11/2009–3/15/2011); "CSR: Medium: Collaborative Research: Scaling the Implicitly Parallel Programming Model with Lifelong Thread Extraction and Dynamic Adaptation" (NSF Award #CNS-0964328, Award Dates: 5/1/2010–4/30/2014); "SPARCHS: Symbiotic, Polymorphic, Autotomic, Resilient, Clean-slate, Host Security" (Columbia 1 GG001705 (PRIME DARPA FA8750-10-2-0253), Award Dates: 9/22/2010–9/21/2014); and, "SI2-SSI: Accelerating the Pace of Research through Implicitly Parallel Programming" (NSF OCI-1047879, Award Dates: 10/1/2010–9/30/2015).

Contents

Abstract iii

Acknowledgments v

List of Tables xii

List of Figures xiii

1 Introduction **1**

1.1 The Need for Automatic Thread Extraction 1

1.2 Speculative Automatic Thread Extraction 4

1.3 Stronger Analysis from a Diversity of Logics 7

1.4 Compiler Scalability 11

1.5 Dissertation Contributions 14

1.6 Assembling an Infrastructure 15

1.7 Dissertation Outline 17

2 Background on Dependence Identification **18**

2.1 Control Dependence 19

2.2 Data Dependence 21

2.2.1 Observable Program Behavior and Side-Effects 22

2.3 Loop-Carried vs Intra-Iteration Dependence 23

2.4 Dependence Graphs 24

2.4.1 Equivalence, Communication, and Synchronization 25

2.4.2 Pipeline Execution, Dependence Cycles, and the $\mathrm{DAG}_{\mathrm{SCC}}$ 25
2.5 Dependence Analysis 26
2.5.1 Demand-driven: Algorithm and Interface 27
2.6 Speculative Dependence Identification 28
2.7 A Brief Overview of the LLVM IR 30

3 The Collaborative Dependence Analysis Framework 32

3.1 Background 33
3.1.1 Example: Array of Structures 34
3.1.2 Example: Unique Access Paths 36
3.1.3 Partiality and Algorithmic Diversity 37
3.1.4 Decomposition and Multi-Logic Queries 39
3.1.5 Combining Analysis Implementations 41
3.2 Structure of an Analysis Implementation in Isolation 42
3.3 Informal Semantics of the Query Language 44
3.4 Foreign Premise Queries, Topping, and Ensembles 45
3.4.1 Example: Solving a Mixed-Logic Query with AoS and UAP 48
3.5 Scheduling Priority 53
3.5.1 Ensuring Termination 54
3.6 Analysis Implementations 55
3.7 Formal Semantics 56
3.7.1 The Instrumentation Semantics 58
3.7.2 Feasible Paths and Loop-Restrictions on Paths 63
3.7.3 The `modref_ii`$(i_1, \texttt{Same}, i_2, H)$ Query 65
3.7.4 The `modref_ii`$(i_1, \texttt{Before}, i_2, H)$ Query 65
3.8 Discussion 67
3.8.1 Development of Factored Analysis Algorithms 67
3.8.2 Generalization to other Analysis Problems 70

3.8.3 Marrying Dependence Analysis with Speculation 71

4 The *Fast* $\mathrm{DAG}_{\mathrm{SCC}}$ Algorithm 72

4.1 Background 73

4.2 Baseline Algorithm 76

4.3 Client-Agnostic Algorithm 77

4.4 Extensions for PS-DSWP 80

4.5 Proof of Correctness 83

4.6 Engineering Considerations 89

4.6.1 Compact Representation of the Set of Vertices 89

4.6.2 Compact Representation of Edges 90

4.7 Discussion 92

4.7.1 Determinism 92

4.7.2 Antagonistic Graphs 93

4.7.3 Integrating Speculation 95

5 Speculative Dependence Identification 96

5.1 Background 97

5.2 Design Constraints and Design Rationale 98

5.3 The Speculation-Module Pattern 99

5.3.1 Speculation Manager 102

5.3.2 Speculative Dependence Analysis Adapter 103

5.3.3 Validation Generator 106

5.3.4 Runtime Support Library 107

5.4 Composability 107

5.4.1 Mechanism, not Policy 108

5.4.2 Instrumentation over Replacement 108

5.4.3 Idempotency 109

5.5 Implementations of Speculation . . . 110
5.5.1 Control Speculation . . . 110
5.5.2 Loop-Invariant Loaded-Value Prediction . . . 111
5.5.3 Memory Flow Speculation . . . 112
5.5.4 Separation Speculation . . . 115
5.5.5 Pointer-Residue Speculation . . . 123
5.6 Discussion . . . 125
5.6.1 Speculative Assumptions with Efficient or Scalable Validation . . . 125
5.6.2 Speculation without Profiling . . . 126

6 Evaluation **128**
6.1 The Collaborative Analysis Framework . . . 128
6.1.1 Importance of Analysis to Speculative Parallelization . . . 129
6.1.2 Absolute Precision . . . 130
6.1.3 Collaboration and Orthogonality . . . 137
6.2 The Fast $\mathrm{DAG}_{\mathrm{SCC}}$ Algorithm . . . 141
6.2.1 Performance Improvement . . . 143

7 Conclusion and Future Directions **149**
7.1 Summary and Conclusions . . . 149
7.2 Future Directions . . . 151
7.2.1 Further Formalization of Dependence Analysis . . . 151
7.2.2 Tools to Aid Development of New Factored Analyses . . . 151
7.2.3 Efficiently Validated Speculative Assumptions . . . 152
7.2.4 Speculation without Profiling . . . 152

A Analysis Implementations **153**
A.1 Theme: Lift May-Alias to May-Depend . . . 154
A.2 Theme: Conservativism . . . 155

A.3 Theme: You Cannot Guess an Address 155
A.4 Theme: Simpler Data Flow on Non-Captured Storage 156
A.5 Auto-Restrict 157
A.6 Basic Loop 157
A.7 Φ-maze 158
A.8 Pure and Semi-Local Functions 158
A.9 Kill Flow 159
A.10 Callsite Depth-Combinator 161
A.11 Global Malloc 162
A.12 Non-captured global 163
A.13 Non-Captured Source 164
A.14 Unique Access Paths 164
A.15 Array of Structures 165
A.16 Scalar Evolution 166
A.17 SMTAA 166
A.18 Sane Typing 167
A.19 Non-Captured Fields 169
A.20 Acyclic 171
A.21 Disjoint Fields 173
A.22 Field Malloc 173

List of Tables

3.1 Summary of Query Language 46

3.2 Summary of Analysis Implementations. 56

6.1 Observed Collaboration, Orthogonality, and Anti-Collaboration. 138

6.2 Hot Loops from SPEC CPU2006 142

List of Figures

1.1 Architectures are more parallel, yet sequential applications do not benefit. . 2
1.2 Results from a 2011 Survey of Princeton University Scientists. 3
1.3 A linked representation of a Matrix. 8
1.4 Regular Traversal of the Linked Matrix Representation 9
1.5 One dependence query requires two dependence logics. 10
1.6 PDG, SCCs and Condensation Graph 13
1.7 The Infrastructure's Training and Planning Phases. 16
1.8 The Infrastructure's Transformation Phase. 17

2.1 Paths of execution, resource footprints, and aliasing pointers. 20
2.2 Speculative Assumptions vs. Dependences 29

3.1 Non-captured Pointers and Points-to Sets 35
3.2 Regions of Precision. 38
3.3 Multi-logic Queries. 40
3.4 Structure Traditional, Best-of-N, and Collaborative Analysis Algorithms. . 43
3.5 Internal organization of a typical dependence analysis algorithm. 44
3.6 Combining forward and reverse `modref_iis` into *may-depend* 47
3.7 Combining Analysis Algorithms into an Ensemble. 49
3.8 Listing for a AoS-UAP Multi-Logic Query 50
3.9 Semantics for Memory Operations 62

3.10 Semantics for branch instructions 63
3.11 The simple multistep relation 65
3.12 Path-restricted small-step relation 65
3.13 The *connects-to-in* relation 66
3.14 Restricting the multistep relation to loops 67
3.15 Semantics of `modref_ii`$(i_1, \texttt{Same}, i_2, H)$ 68
3.16 Semantics of `modref_ii`$(i_1, \texttt{Before}, i_2, H)$ 69

4.1 Redundant Edges in the PDG 75
4.2 Constructive Edges in the PDG 75
4.3 Algorithms `withTheGrain`, `againstTheGrain` use topological ordering to discover constructive edges. 78
4.4 Eight bits characterize the dependences between two vertices. 90
4.5 Sorted adjacency list PDG representation. 91

5.1 Speculation Managers and Dependence Analysis Adapters in the Compiler's Planning Phase 101
5.2 Validation Generators in the Compiler's Transformation Phase 101
5.3 Validation of Loop-Invariant Loaded-Values 113
5.4 Accumulator expansion 119
5.5 Pointer residues. 124

6.1 Analysis Precision affects Validation Overheads and Impacts overall Performance. 131
6.2 Context Improves PDG Client's Precision 133
6.3 Context Improves PS-DSWP Client's Precision 134
6.4 Context Reduces PS-DSWP Client Bail-outs 135
6.5 Collaborative vs. No-topping composition. 137
6.6 How to measure collaboration, orthogonality, and anti-collaboration. 138

6.7 Measured Collaboration. 139
6.8 Fast $\mathrm{DAG}_{\mathrm{SCC}}$ running time strongly correlated to reduction in Queries . . . 145
6.9 Largest sequence of hot loops analyzed before timeout. 146
6.10 Per-loop $\mathrm{DAG}_{\mathrm{SCC}}$ progress on SPEC CPU2006 benchmarks. 148

A.1 Callsite depth-combinator's search 162

Chapter 1

Introduction

"If I'd asked customers what they wanted,

they would have said 'a faster horse.'"

—Henry Ford.

This dissertation presents a middle-ware for automatic parallelization. This infrastructure features novel components spanning dependence analysis and thread extraction transformations, and has proved flexible and general enough to support many research projects [36, 37, 38, 45, 65, 69, 57]. A primary motivation for this work is automatic thread extraction, wherein this infrastructure identifies (or creates) independence among program statements and schedules them for concurrent execution. Combined, these elements contribute to drastic performance improvements by extracting threads from sequential, general-purpose applications.

1.1 The Need for Automatic Thread Extraction

Presently, the microprocessor industry invests in increasingly parallel architectures rather than improvements to sequential application performance. Before the "multicore era," application developers could rely on advancements in micro-architecture to deliver a steady

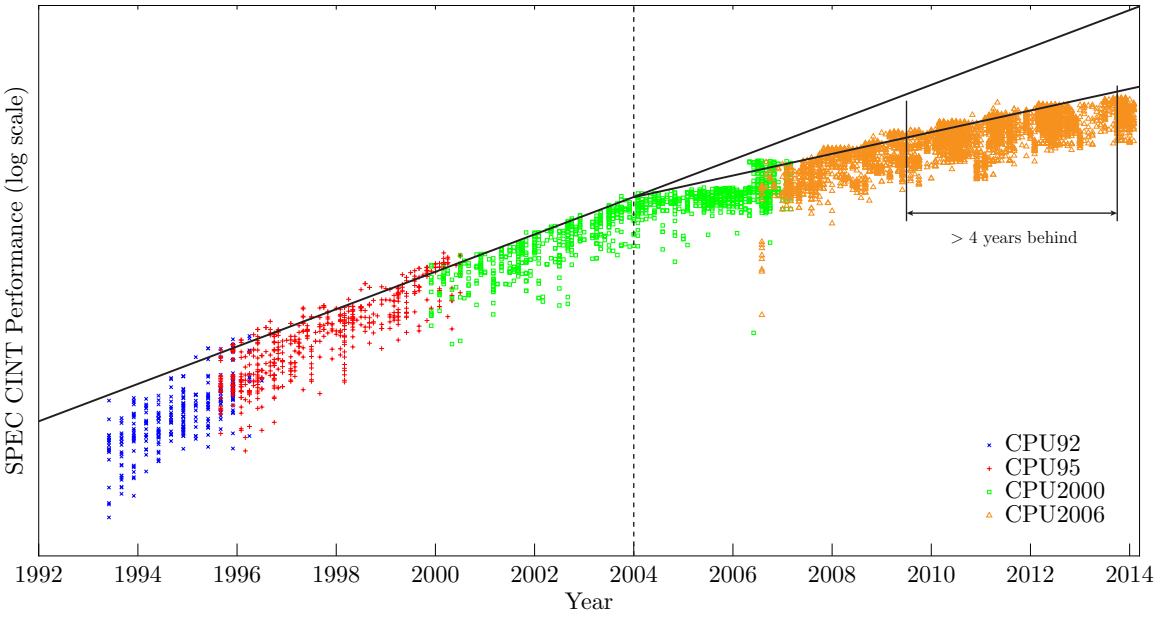
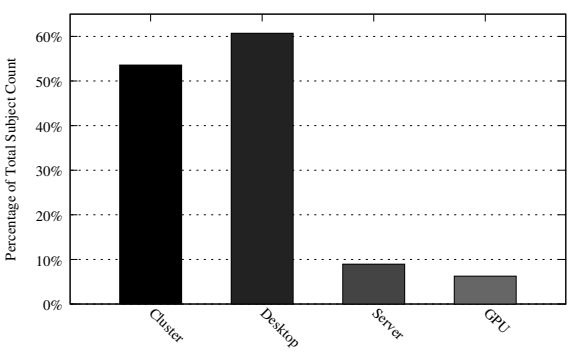


Figure 1.1: Performance results for the SPEC CPU92, CPU95, CPU2000, and CPU2006 benchmark suites over the last 20 years as reported on the SPEC website [78]. Architectures are more parallel yet sequential applications do not benefit.

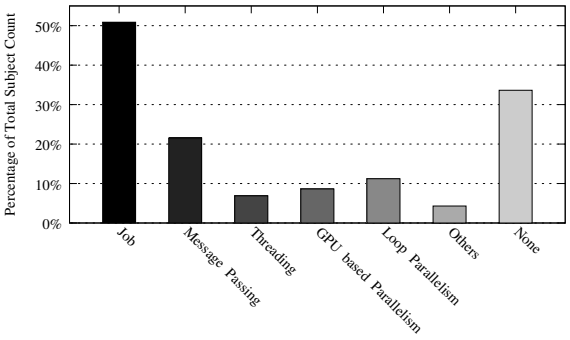
performance improvement. Although multicore processors provide additional computational resources, sequential applications do not benefit from these resources unless they are restructured for parallelism.

Figure 1.1 illustrates the difference caused by multicore through the reported performance on the SPEC benchmark suites over the last twenty years. Performance (vertical axis) is normalized log-scale. Micro-architectural improvements deliver exponential performance improvement. However, that growth does not continue at the same rate after 2004, coinciding with multicore architectures. Although newer architectures provide more parallel resources, they do not deliver the same performance scaling unless applications are restructured for parallelism.

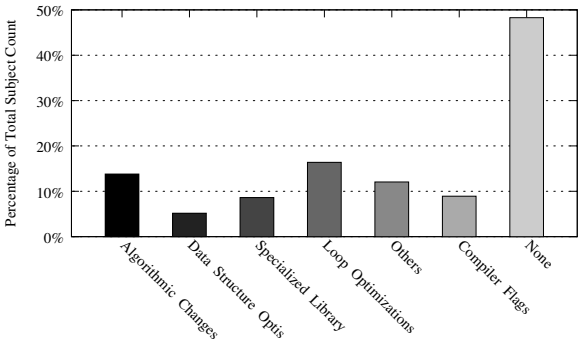
In 2011, the Liberty Research Group surveyed 114 computational scientists from Princeton University [66]. The results from this survey confirm that architectures commonly used by computational scientists are parallel. Figure 1.2a shows that more than half of the computational scientists routinely use clusters in their research. More than half of the respon-



(a) Types of architectures employed.



(b) Types of parallelism employed.



(c) Common performance optimizations employed.

Figure 1.2: Results from a 2011 Survey of Princeton University Scientists [66].

dents use desktop computers, most of which include multicore processors. The community has a great deal of parallel resources.

Nearly half of surveyed scientists wait days for program completion and 15% wait months. These researchers would benefit from faster computation; 85% reported that faster computation would "profoundly change" the way they do research. Nonetheless, nearly half of researchers perform no optimization of their codes (Figure 1.2c), and more than 30% of researchers do not use parallelism (Figure 1.2b), saying that existing abstractions for parallelism are "hard," "look complex," or have "big learning curves" [66]. Indeed, all systems which rely on explicit parallelization (via annotations or library-level primitives) have a learning curve which distracts from the main goals—application correctness and feature development.

1.2 Speculative Automatic Thread Extraction

A promising alternative to explicit parallelism is the extraction of threads via an automatic parallelization system. At a high level, automatic parallelization addresses two problems: identifying independence among the operations in a Program-Under-Optimization (PUO) and scheduling independent work for concurrent execution. This dissertation primarily concerns the identification of independent work through static analysis and enabling transformations such as speculation.

Classical compiler transformations use the results of static analysis to avoid unsound optimizations. For instance, if analysis reports that one operation from the PUO *depends on* another operation, the compiler avoids any transformation which may reorder those operations and thereby violate the dependence. By preserving all dependences, the compiler preserves observable program behavior [34]. However, imprecise static analyses fail to disprove certain spurious dependences, and the compiler conservatively limits transformation accordingly. The quality of analysis directly limits the compiler's freedom to transform the code, encouraging the development of more precise analysis algorithms.

Improving the precision of static analysis increases the number of sound program transformations available to the compiler [15, 27, 71, 74, 88]. However, this approach has limitations. Static may-alias analysis (a building block of dependence analysis) is undecidable [49]: although a dependence analysis algorithm may be *precise enough* to support particular sound optimizations on a fixed set of PUOs, there will always be a counterexample PUO for which the algorithm is so imprecise as to inhibit an otherwise-sound optimization. Second, since static analysis has no knowledge of the PUO's intended execution environment (e.g. its run time input set), static analysis must report conservative results which generalize across *all* program behaviors induced by *any* execution environment. These generalized results prevent optimizations that are sound for realistic program inputs because the optimizations are unsound for some antagonistic input that never occurs in practice. Consequently, compilers which rely solely on static analysis fail to extract threads from a

broad range of PUOs.

Speculative optimization has emerged in response to these limitations [20, 40, 45, 46, 58, 68, 76, 81, 93]. Through speculation, an optimization system simplifies static analysis problems by making *assumptions* about the PUO. Under these assumptions, certain worst-case program behaviors are impossible, allowing *speculative dependence analysis* to report *optimistic* results which more precisely reflect expected-case program behavior and thus grant the optimizer additional freedom. To preserve program behavior, *speculative transformation* generates additional code to *validate* those assumptions during *speculative execution* and signal *misspeculation* if those assumptions fail. Speculative execution *recovers* from misspeculation by *rolling back* program state and *re-executing* code which does not include optimizations based on the faulty assumption.

Speculative optimizations improve applicability by re-casting transformation soundness as a performance concern. The net running time of a speculative execution is a mixture of two cases: the assumptions hold or they do not. For sake of discussion, net running time is approximated as an affine combination of these cases:

$$\begin{aligned}(\text{Net Time}) &= (1 - \text{Misspec Rate}) \times (\overbrace{\text{Checkpoint} + \text{Validation}}^{\textit{Overhead}} + \text{Optimized Time}) \\ &+ (\text{Misspec Rate}) \times (\underbrace{\text{Roll Back} + \text{Re-Execution Time}}_{\textit{Overhead}})\end{aligned}$$

Speculation is a net win when the improvement due to optimization outweighs the overheads. Modern speculative optimization systems pursue *high-confidence speculation* to minimize the misspeculation rate and effectively eliminate the overheads of rollback and non-speculative re-execution. Several proposals use the operating system's copy-on-write facility to reduce checkpointing overheads to insignificance [45, 47, 68]. Even if misspeculation never occurs, speculative execution incurs validation overheads in the common case.

Certain designs of speculative transformation impose high validation overheads which

are difficult to eliminate [13, 45]. Speculative parallelization often uses transactional memory systems to achieve validation and recovery. In transactional memory systems, validation must observe the actual sequence of memory updates performed by all transactions to determine whether loads within each transaction see a value consistent with a valid sequential execution. The parallelization system emits additional instructions into the parallelized application which observe every `store` and certain `load` instructions so they may be replayed to discover conflicts [13, 45, 47, 68].

A conservative quantitative estimate of validation overheads places the overheads of such transactional memory systems in perspective. The replay operations can be offloaded to another core [45, 47, 68], thus only those instructions which were inserted to observe and communicate memory accesses slow the progress of a speculative worker process. On commodity hardware, these communications can be implemented via a queue data structure, and the `enqueue` operation must consist of at least one `store` operation. In other words, an instrumented `store` becomes two `stores` (or worse). If we estimate that 9–13% of dynamic instructions are `store` instructions [30, 64], then validation imposes at least a 9–13% overhead. Separately, these additional `stores` increase memory bandwidth requirements. Even with a small number of worker processes, communications for validation can easily exceed 1 GBps [45]. When validation bandwidth exceeds the hardware's communication bandwidth, speedup is impossible.

Although speculative dependence analysis frequently enables thread extraction, the overheads of transactional validation may negate performance gains from concurrency. Stronger analysis can eliminate or reduce those overheads either by allowing non-speculative parallelization or by reducing the number of speculative assumptions needing validation. Thus, transactional validation benefits tremendously from precise static analysis.

1.3 Stronger Analysis from a Diversity of Logics

Program analysis algorithms uncover important facts about an input program. These facts drive compiler optimization, bug finding tools, and many other applications. Decades of research have uncovered a broad array of analysis algorithms. For instance, algorithms for *pointer analysis* (including points-to analysis and alias analysis) judge whether two pointers within a program may reference the same memory location [4, 10, 11, 52, 55, 80, 91]. Algorithms for *shape analysis* (including heap reachability analysis) model how a program links its memory objects into data structures in order to answer questions of disjointness or cyclicity of data structures as a whole [26, 28, 77]. These shape analysis facts may answer certain classes of pointer aliasing, in turn. Algorithms for loop dependence analysis determine whether memory accesses from different iterations of a loop must happen in sequential program order, or if they can execute out of order [7, 67]. Research continues since these problems are difficult and not solved; the general cases are undecidable, and various abstractions are decidable though intractable [33, 60]. Each proposal is an approximation and occupies a distinct niche in the trade-off between precision and scalability [31], but no algorithm dominates all others.

Each analysis algorithm represents one logic for dependence reasoning. Each logic precisely recognizes a certain restricted case of dependence analysis but yields imprecise results in other cases. These restricted cases often have an interpretation in the source language. For example, the Omega Test compares pointers that are *affine functions of loop induction variables* [67]. However, developers compose disparate language features into their programs, resulting in dependence queries which exceed any one analysis' region of precision. To analyze such programs, several logics are necessary.

More concretely, consider an example code which manipulates a non-trivial data structure. Figure 1.3 demonstrates how the "array-of-row-vectors" `Matrix` data structure adheres to a simple acyclic shape. Figure 1.4 demonstrates code which traverses and updates the matrix according to a regular (affine) iteration pattern.

```
1  // Array of row vectors.
2  typedef struct
3    { float cells[M]; } Row;
4
5  typedef struct
6    { Row *rows[N]; } Matrix;
```

```
7  // Create a new Matrix
8  Matrix *new_matrix() {
9    Matrix *m =
10     malloc( sizeof( Matrix ) );
11
12   // each row a separate object
13   for(int i=0; i<N; ++i)
14     m->rows[i] =
15       malloc( sizeof( Row ) );
16   return m;
17 }
```

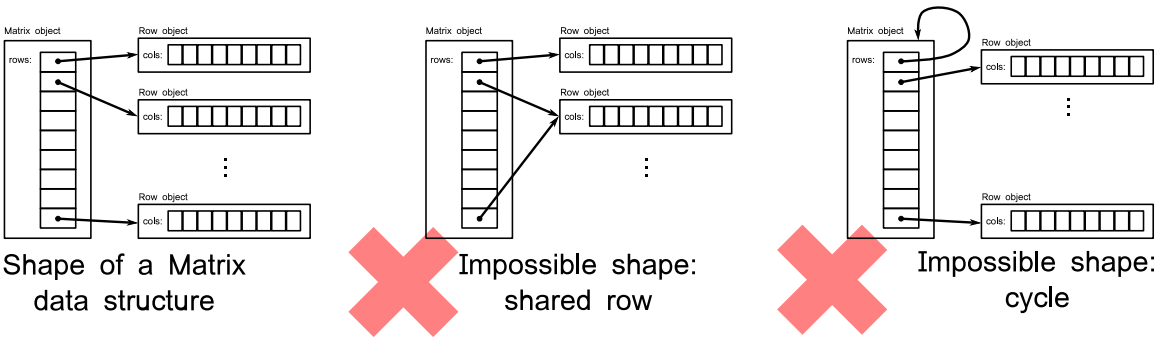


Figure 1.3: *(above)* Type definition and initialization routine for a linked, "array-of-row-vectors" representation of an $N \times M$ Matrix; *(bottom)* Such `Matrix` structures take only certain "shapes" in any program execution.

```
18 // Inputs: m is an NxM Matrix; I,J is a fixed cell in m.
19 // Algorithm: let m[I,J] = min_{J<k<M} m[I,k]
20 // Row, Matrix defined in Figure 1.3.
21
22 // load 1:
23 Row *tmp1 = m→rows[I];
24 // load 2:
25 float cij = tmp1→cells[J];
26
27 for(int k=J+1; k<M; ++k) {
28   // load 3:
29   Row *tmp2 = m→rows[I];
30   // load 4:
31   float cik = tmp2→cells[k];
32
33   if( cij > cik ) {
34     // load 5:
35     Row *tmp3 = m→rows[I];
36     // store 1:
37     tmp3→cells[J] = cik;
38     cij = cik;
39   }
40 }
```

Figure 1.4: Example code traverses a linked Matrix data structure in a regular pattern. It updates matrix cell $m[I, J]$ with the minimum element among $m[I, k]$ for $J < k < M$.

To analyze this example code, the compiler must determine whether there is a dependence (flow of information) among its memory accesses. In particular, Figure 1.5 shows a proof that there is no flow dependence from `store 1` to `load 4` across iterations of the inner loop. Through two sub-goals, the proof argues that the pointers cannot alias. The first sub-goal establishes that `store 1` does not alter any pointers to `Row` objects—i.e. that the shape of the matrix data structure is invariant—and is built from a heap-reachability argument [77, 26, 28]. The second sub-goal, which establishes that the loop iterates monotonically across one row, is built from Linear Integer Arithmetic (LIA). Both logics are required for this one dependence query. Although there are decision procedures for LIA [67], the first sub-goal cannot be proved within LIA.

Claim: There is no dependence from `store 1` (line 37) to `load 4` (line 31) across iterations of the inner loop (line 27).
Proof: *(by disproving aliasing)*

1. Sub-goal: $\texttt{tmp1} = \texttt{tmp2}$ and $\texttt{tmp1} = \texttt{tmp3}$.

   (a) Initially, each element of $\texttt{m} \rightarrow \texttt{rows}$ refers to a `Row` object (line 14).

   (b) No store instruction in the loop mutates any `Matrix` object; the `Matrix` object is loop-invariant. *(Although* `store 1` *mutates a* `Row` *object (line 14), those are disjoint from* `Matrix` *objects (line 9)).*

   By (a) and (b), `load 1`, `load 3` and `load 5` observe the same invariant element of $\texttt{m} \rightarrow rows$.

   Thus $\texttt{tmp1} = \texttt{tmp2}$ and $\texttt{tmp1} = \texttt{tmp3}$.

2. Sub-goal: $\texttt{tmp2} \rightarrow \texttt{cells}[\texttt{k}] > \texttt{tmp3} \rightarrow \texttt{cells}[\texttt{J}]$.
   (by induction on k)

   Base $\texttt{k}_0 = \texttt{J} + 1$, thus,
   $\texttt{tmp2} \rightarrow \texttt{cells}[\texttt{k}_0] = \texttt{tmp1} \rightarrow \texttt{cells}[\texttt{J} + 1]$
   $> \texttt{tmp1} \rightarrow \texttt{cells}[\texttt{J}] = \texttt{tmp3} \rightarrow \texttt{cells}[\texttt{J}]$.

   Ind. $\texttt{k}_{i+1} = 1 + \texttt{k}_i$, thus,
   $\texttt{tmp2} \rightarrow \texttt{cells}[\texttt{k}_{i+1}] > \texttt{tmp2} \rightarrow \texttt{cells}[\texttt{k}_i]$
   $> \texttt{tmp3} \rightarrow \texttt{cells}[\texttt{J}]$.

By (2), `store 1` and `load 4` must access non-aliasing pointers, thus there is no dependence. □

Figure 1.5: A proof corresponding to a negative result of a non-trivial dependence query on operations from the example code. This proof employs two types of reasoning: the first sub-goal relies on an heap reachability argument and the second sub-goal relies on a Linear Integer Arithmetic (LIA) argument.

To address *both* sub-goals, we need logics for each and a means to combine them. One means to combine these two logics would be to build a rich model of the analysis problem which supports both types of reasoning, and then design procedures which apply those logics to the combined model. However, this quickly becomes unwieldy since each additional analysis logic must be considered in relation to all others. Instead, this dissertation proposes a design whereby each analysis implementation is restricted to one logic, accepting that some premises of a query cannot be analyzed within that specific logic. Those simple implementations combine through a property called *collaboration*. Two algorithms collaborate if—while maintaining composability—the combination disproves dependences which neither algorithm disproves in isolation. Each analysis algorithm may then recruit other analysis algorithms to solve their *foreign premises*.

Once collaboration is established, developers may modularize the development of analysis algorithms through *factorization*. Instead of increasingly complicated algorithms which incorporate additional types of reasoning, factorization achieves precision through many simple algorithms. Each algorithm disproves queries within its core competence and assumes other algorithms provide the necessary diversity of logic to solve its premises. Factored algorithms are developed independently without requiring knowledge of others. Factorization enables developers to easily extend algorithm precision according to the needs of a client.

1.4 Compiler Scalability

Users desire compilers to be fast; one study [83] indicates that programmer productivity drops when compilation takes more than a few seconds. If the benefit of aggressive optimizations does not outweigh the cost of long compile times, users will avoid those optimizations. This effect is observed in the development of real-world compilers: the GCC Development Mission Statement lists “faster debug cycles” as one of its six design and

development goals [25], and the GCC manual states that GCC will "refuse to optimize programs when the optimization itself is likely to take inordinate amounts of time" [23].

However, several aspects of aggressive optimization dilate compilation time. Analysis precision drastically affects optimization quality [15, 27, 71, 74, 88], and precise analyses tend to be more expensive than their less-precise counterparts [22, 32] and scale poorly [33, 60]. Similarly, Amdahl's Law [3] suggests that the benefit of advanced compiler optimizations is greatest when applied to *hot* program scopes. Such hot scopes often span large regions of program execution, thus exaggerating the scalability of a compiler's underlying algorithms. Consequently, aggressive optimizing compilers tend to run slowly, making them a less appealing tool for developers. Despite the performance potential of state-of-the-art transformations, common compilers optimize small, intra-procedural scopes, instead favoring short compilation times. If optimization takes too much time, developers evict it from their development cycle.

To extend the benefits of aggressive optimization to the wider community, we must first address scalability of precise analysis. We envision a future where common compilers feature aggressive optimizations such as automatic parallelization [17, 58, 71, 74, 76, 84, 88] by default. The critical path to this end is the precise analysis of large program scopes.

Many compiler techniques are formulated around the Program Dependence Graph (PDG) [21] (Figure 1.6(a)). Many of those techniques (*clients* of the PDG) focus primarily on dependence cycles, identified as the Strongly Connected Components (SCCs) of the PDG (Figure 1.6(b)).

The Directed Acyclic Graph of the SCCs ($\mathrm{DAG_{SCC}}$) or *condensation* of the PDG is a representation that makes dependence cycles explicit (Figure 1.6(c)). The $\mathrm{DAG_{SCC}}$ contains enough information to support a broad class of compiler techniques. For instance, automatic parallelization [17, 58, 71, 74, 76, 84, 88] must determine whether two operations can execute concurrently without races, or if synchronization is needed. The $\mathrm{DAG_{SCC}}$ conveys this relationship: synchronization is necessary if those operations are assigned to

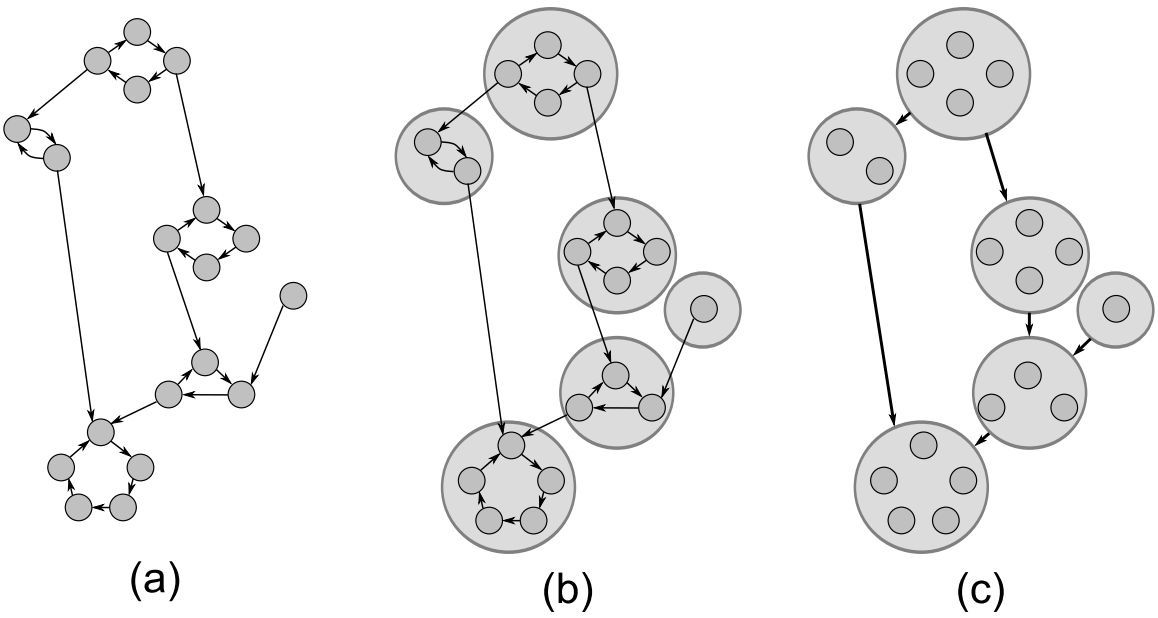


Figure 1.6: (a) Example PDG; (b) Strongly Connected Components; (c) Condensation of the example.

the same component or if their components are ordered with respect to one another. Similarly, program slicing tools [35, 90, 92] report a "backwards program slice" of an operation by enumerating those operations assigned to the same component[1] as the operation of interest as well as operations assigned to components ordered before it. in the $\mathrm{DAG}_{\mathrm{SCC}}$. The loop fission transformation splits a loop into two or more parts [6, 43], and is valid when it preserves components and the ordering of components visible in the $\mathrm{DAG}_{\mathrm{SCC}}$.

The $\mathrm{DAG}_{\mathrm{SCC}}$ holds less information than the PDG and should be cheaper to compute. Yet, standard practice wastefully builds the full PDG before condensing it to a $\mathrm{DAG}_{\mathrm{SCC}}$. The number of potential PDG edges grows quadratically with the scope size (in vertices). Each potential edge adds a quantum of analysis effort (a *query*) to determine whether that edge exists. The running times of these queries sum to make $\mathrm{DAG}_{\mathrm{SCC}}$ construction prohibitively expensive, especially since precise analyses are costly [22, 32, 33, 60].

Compiler authors should not sacrifice analysis precision for cost since imprecision limits optimization [15, 27, 71, 74, 88]. Instead, they should use the most precise analyses and

[1] We use *component* to refer to a strongly connected component; in contrast, component refers to a program statement in the program slicing literature.

reduce compilation time by exploiting the reduced information of the $\mathrm{DAG}_{\mathrm{SCC}}$.

This dissertation presents a technique that computes the $\mathrm{DAG}_{\mathrm{SCC}}$ more efficiently than finding SCCs of the full PDG. Using partial dependence information, the algorithm identifies dependence edges which cannot affect the clients of the $\mathrm{DAG}_{\mathrm{SCC}}$. Next, the algorithm uses a Demand-Driven [29, 79, 96] analysis framework to elide those analysis queries and thus expend effort only on important analysis queries rather than the whole program. This improvement is orthogonal to reducing the latency of each query; it reduces $\mathrm{DAG}_{\mathrm{SCC}}$ construction time yet maintains high analysis quality since no analysis algorithms change. With these savings, compiler authors may pursue more aggressive and costlier analyses while providing the same quality of service to compiler end-users.

1.5 Dissertation Contributions

These points represent the largest contributions of this dissertation:

Collaborative Dependence Analysis Framework (CAF): This dissertation first provides a novel means to compose several analysis algorithms into an *ensemble* algorithm which features the strengths of each member while servicing queries as fast as the fastest member in the expected case. Next, this dissertation enhances the composition mechanism to support *collaboration*, allowing several simple analysis algorithms to solve queries which none can solve alone. With collaboration established, one may *factor* analysis algorithms into small, orthogonal pieces to reduce development effort while maintaining the precision of a single all-encompassing algorithm. Chapter 3 describes CAF.

The Fast $\mathrm{DAG}_{\mathrm{SCC}}$ Algorithm: Analyzing larger program scopes presents a scalability challenge. Fortunately, many compiler optimizations are driven by the Strongly Connected Components of the Program Dependence Graph ($\mathrm{DAG}_{\mathrm{SCC}}$) rather than the whole PDG. Since the $\mathrm{DAG}_{\mathrm{SCC}}$ contains less information than the PDG, this dissertation presents an adaptive algorithm to compute the $\mathrm{DAG}_{\mathrm{SCC}}$ using fewer dependence analysis queries. Av-

eraged across the SPEC 2006 suite, this algorithm reduces analysis time by half while maintaining equivalent analysis precision, thus performing aggressive program analysis in a reasonable execution time. Chapter 4 describes the Fast $\mathrm{DAG}_{\mathrm{SCC}}$ Algorithm.

Integrating Speculation: This dissertation presents a design pattern wherein various types of speculation are designed and implemented modularly, and can be *plugged in* to the compiler without modifying other parts of the compilation framework. These implementations of speculation naturally compose with one another, and may collaborate with state analysis in the CAF. Chapter 5 describes how speculation integrates with analysis and transformation in a modular way.

1.6 Assembling an Infrastructure

This dissertation demonstrates an end-to-end integration of these techniques into an automatic parallelization system, and presents a modular design of the claimed contributions (and many other pieces). Pieces from this infrastructure have supported the research needs of the Liberty Group [36, 37, 38, 45, 57, 65, 69].

Figure 1.7 illustrates the training and planning phase of the compiler infrastructure. (1) The developer provides sequential source code and a representative input set. This is compiled to (2) LLVM's intermediate representation. During the training phase (3) one or more profilers instrument the IR and collect dynamic execution information.

The compiler loads those profiling results, and speculation managers (4) interpret the results to identify a set of high-confidence assumptions about program behavior—that is, assumptions which will likely hold true at run time. The system does not yet commit to those assumptions because it does not yet know whether these assumptions are necessary to enable transformation.

The Fast $\mathrm{DAG}_{\mathrm{SCC}}$ algorithm (5) analyzes the program to compute the strongly connected components of the program dependence graph. In this process, all memory de-

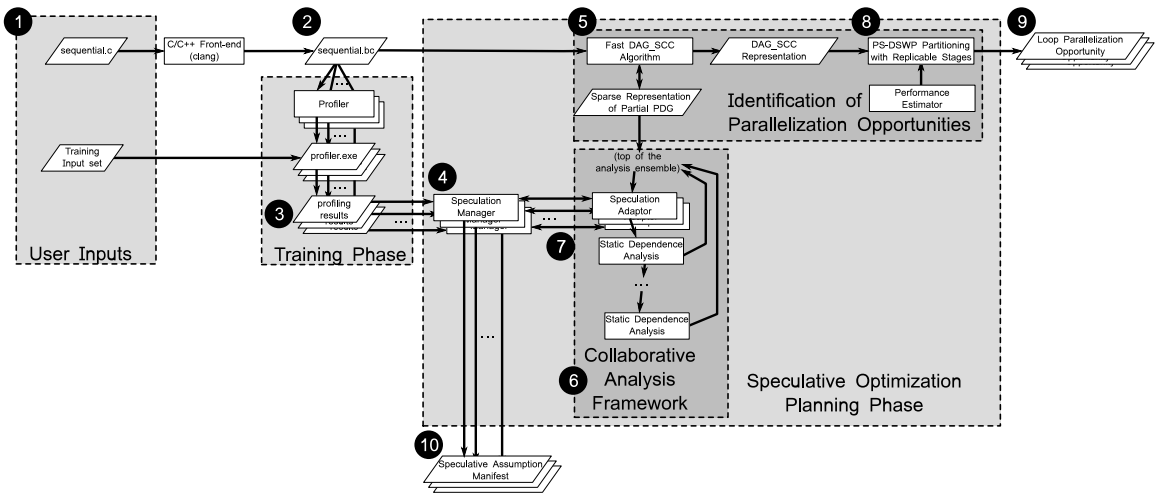


Figure 1.7: The Infrastructure's Training and Planning Phases.

pendences are resolved by the CAF (6). To support speculation, one or more speculation adapters (7) are inserted into CAF. These adapters may report independence according to the high-confidence limits identified by the speculation managers (4), and record which assumptions are *actually* used while analyzing the program into speculative assumption manifests (10).

The PS-DSWP thread partitioning heuristic (8) assigns the SCCs from the $\mathrm{DAG}_{\mathrm{SCC}}$ to pipeline stages while trying to balance those stages to maximize concurrency. Thread partitioning may identify several loop parallelization opportunities (9).

Finally, the speculation managers report speculative assumption manifests (10) which enumerate the speculative assumptions that must be validated to ensure correctness of the parallelization transformation (9).

Figure 1.8 illustrates the second half of the compiler infrastructure: transformation. Earlier passes have identified one or more loop parallelization opportunities in the sequential IR (9) subject to zero or more speculative assumption manifests (10). What remains is to insert validation checks for each type of speculation, and to parallelize the code.

Each speculative assumption manifest (10) corresponds to a different type of speculation, e.g., control speculation, silent store speculation, or transactional serializability. The

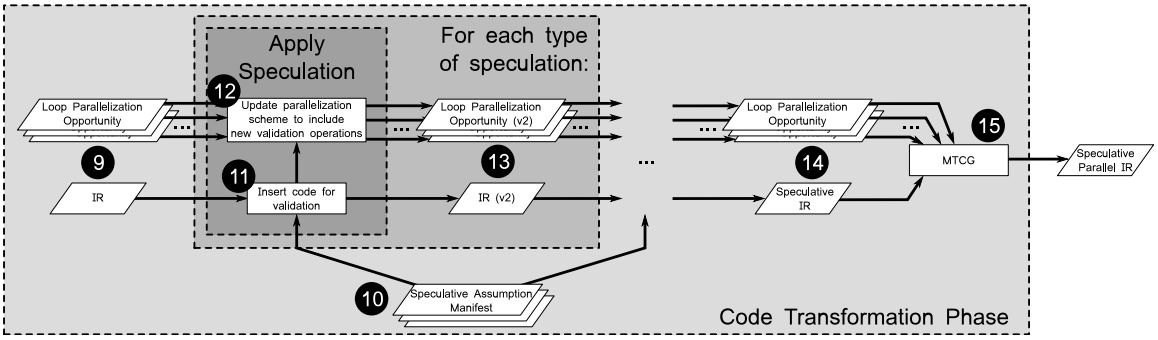


Figure 1.8: The Infrastructure's Transformation Phase.

compiler inserts validation for each in turn. First, it inserts additional instructions to the sequential IR to perform validation of each assumption listed in the manifest (11). Next, it updates the loop parallelization opportunities (12) so that the new validation instructions are assigned to the appropriate pipeline stage at runtime. This results in new versions of the IR and loop parallelization opportunities which are still sequential, yet are speculative (13). After validation has been inserted for all types of speculation, the speculative sequential IR and loop parallelization strategies (14) are fed into the Multi-Threaded Code Generation algorithm [63] (15) to produce a speculative parallel IR. This IR can be lowered to a machine-binary.

1.7 Dissertation Outline

Chapter 2 reviews background information on dependence analysis, dependence graphs, and speculation. Chapter 3 explores CAF (Figure 1.7, point 6). Chapter 4 explores the Fast $\mathrm{DAG_{SCC}}$ Algorithm (Figure 1.8, point 5). Chapter 5 explores the integration of speculation with the framework (Figures 1.7–1.8, points 2–4, 7, and 11–15). Chapter 6 presents an experimental evaluation of these contributions.

Chapter 2

Background on Dependence Identification

"It's much more interesting to live not knowing
than to have answers which might be wrong."
—Richard Feynman.

Through *dependence identification*, an optimization system quantifies its degrees of freedom for rescheduling the operations (statements, instructions, etc.) within a program under optimization (PUO). This chapter introduces the pieces of dependence identification, including the notions of a dependence among a PUO's operations, dependence graphs, and using speculation to safely ignore unlikely dependences.

Our interest in dependences stems from the compiler's goal of preserving observable program behavior through transformation. Dependences constitute an "adequate" representation of program behavior, i.e., provided that optimization respects all dependences in the input program, the optimized output is strongly equivalent to the original [34]. Adequacy established, dependence graphs are additionally a convenient representation for scheduling problems in general and for automatic thread extraction in particular.

Informally, a dependence among operations represents any constraint which would prevent said operations from executing in an order other than source program order, i.e., the order in which they appear in the sequential input code. Two language features introduce these constraints: control flow and data flow. *Control dependences* arise when control flow operations may cause or prevent another operation from executing; for instance, a `for`-loop controls the statements within its body. *Data dependences* arise when one statement computes some value which is used by and affects the operation of another statement.

2.1 Control Dependence

This dissertation adopts the definition of control dependence from [18]:

Definition 1 (Control Dependence among CFG Nodes (Cytron et al. [18]))**.** *Let X and Y be two nodes in a control-flow graph. We say Y is* control dependent on *X iff X is in the* post-dominance frontier *of Y.*

Note that this definition is equivalent to the definition from [21].

Following LLVM [51], this dissertation employs control flow graphs drawn over basic blocks as opposed to individual operations. Operations which have more than one successor in the control-flow graph appear only at the end of basic blocks. Thus, this dissertation lifts the definition of control dependence to accommodate operations in basic blocks:

Definition 2 (Control Dependence among Instructions)**.** *Let x be a multiple-successor operation at the end of basic block X, and let y be an operation in basic block Y. We say that y is* control dependent on *x—or, that x* controls *y—iff Y is control dependent on X.*

Note that the control dependences relation is not symmetric, reflexive, nor transitive.

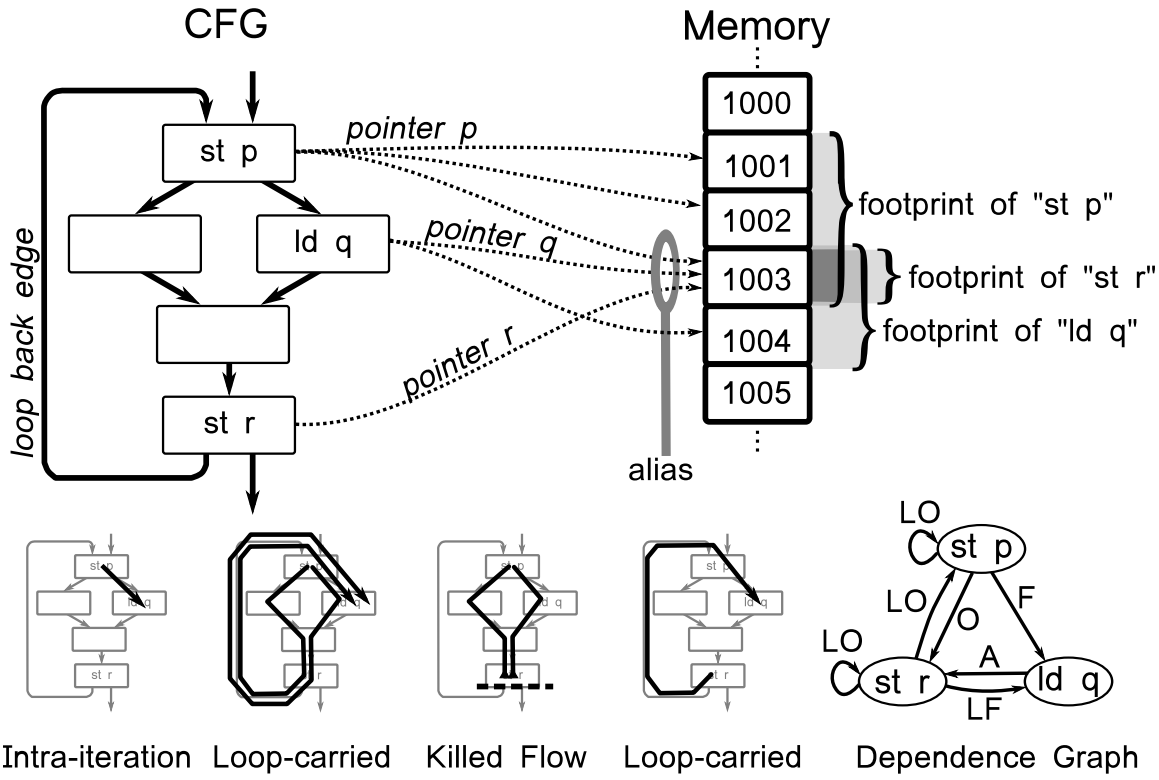


Figure 2.1: (*top*) Loads (`ld`) and stores (`st`) access memory via pointers. Pointers *alias* if they reference a common location. A *footprint* lists resources which an operation accesses. (*bottom*) Intra-iteration path from `st p` to `ld q`; Some loop-carried paths from `st p` to `ld q`; Operation `st r` kills loop-carried flows from `st p` to `ld q`; but, operation `st p` does not kill all loop-carried flows from `st r` to `ld q`; *(F)low*, *(A)nti*, and *(O)utput* dependences, *(L)oop-carried*.

2.2 Data Dependence

Data dependences represent execution ordering constraints due to the flow of values among operations or due to the reuse of storage resources. This dissertation employs the following definition of data dependence.

Definition 3 (Data Dependence)**.** *Let t, u be two operations and M a storage location. We say there is a data dependence from t to u iff*

1. *(*alias*) both t and u read or write M;*
2. *(*update*) at least one of t or u writes M; and*
3. *(*path*) there is a feasible path of execution P that visits t before u such that,*
4. *(*no kill*) no operation between t and u in P overwrites M.*

To emphasize M, we sometimes say there is a dependence from t to u via M.

Note that the data dependence relation is not symmetric, reflexive, or transitive. Figure 2.1 illustrates all the key parts of this definition, including pointer aliases, paths of execution, and killed flows.

Data dependences are further classified according to the cause of that constraint.

- *Flow* dependences—also "true" or "read-after-write"—relate an operation which writes ("defines," "updates," "assigns," "mutates," or "stores") a value to any operation which reads ("uses," "inspects," or "loads") that value. We will sometimes say t *flows (through M)* to u meaning that there is a flow dependence from t to u (through the shared resource M).
- *Anti* dependences—also "write-after-read"—relate an operation which reads a value to subsequent operations that overwrite said value.
- *Output* dependences—also "write-after-write"—relate an operation which writes a value to subsequent operations that overwrite said value.

Note that condition 2 of Definition 3 excludes "read-after-read" or "input" dependences. Most clients ignore read-after-read dependences since few memory models experience an observable effect upon reading memory.

Most authors mark a distinction between data dependences carried via registers[1] (*register dependences*) and those carried through memory (*memory dependences*). This distinction corresponds to the worst-case hardness of problems to compute dependences among operations which manipulate storage locations in these classes. Simple and complete analyses—such as def-use and use-def chains [5] or the more efficient Static Single-Assignment (SSA) form [18]—conservatively summarize flow dependences through registers. However, indirect reference allows non-obvious accesses to a storage location, thus necessitating deeper analysis to conservatively account for all accesses.

This dissertation assumes that the compiler's intermediate representation is in SSA form [18], hence computing register data dependences is trivial. Registers in SSA cannot induce output dependences since each register has exactly one definition. Registers in SSA cannot induce anti dependences since register definitions dominate all uses.

2.2.1 Observable Program Behavior and Side-Effects

Beyond branching and accessing memory, programs may issue system calls to achieve effects which are visible outside the program's execution context. Compiler transformation should not re-order such side-effects. This suggests the need for some form of *side-effect* dependence to order side-effecting operations. Observing that the adequacy of dependence graphs [34] is proved over memory states in a programming model that lacks side-effects, this dissertation models side-effect dependences as memory dependences.

[1] Here, "register" denotes a storage location that must be accessed directly through a unique and consistent name, i.e., pointers cannot access the location indirectly. This is consistent with machine registers on most architectures and with *virtual registers* in most compiler intermediate representations.

2.3 Loop-Carried vs Intra-Iteration Dependence

Until this point, dependence has been discussed as a relation among the static operations within a program under optimization. However, those static instructions may represent several dynamic instances during program execution, and it is sometimes important to consider dependences among some yet not all dynamic instances of a static instruction. The dynamic instances created via loop iteration are of particular importance to thread extraction techniques, and there are many ways to disambiguate those instances.

The most expressive representation of dynamic instances, conceptually, is to completely unroll the loop as to consider each dynamic instance separately. Iteration dependence graphs [7] draw dependences among dynamic iterations of a loop, or among the dynamic instances of each operation corresponding to each loop iteration. However, these representations are difficult to compute when iteration bounds cannot be determined statically.

Dependence distance and direction vectors [7] exploit regularity and symmetry to succinctly represent the dependence between two operations in a loop nest. Distance vectors have one distance element d_i corresponding to each enclosing loop L_i, indicating that the dependence occurs every d_i-th iteration of L_i. Although expressive, distance and direction vectors are generally limited to loops with regular iteration patterns and are difficult to employ for general purpose applications.

This dissertation employs a simplification of dependence distances; given a loop L, a dependence may be *intra-iteration* (zero distance) or *loop-carried*[2] (non-zero distance) with respect to L [63, 71, 88]. The bottom of Figure 2.1 illustrates loop-carried and intra-iteration paths. Although less expressive than dependence distance, the loop-carried classification is more easily recognized in irregular PUOs yet is still powerful enough to support thread extraction techniques. More formally,

Definition 4 (Loop-carried, Intra-iteration Dependences)**.** *A dependence from* t *to* u *is* loop-

[2]Loop-carried dependences are also called *inter-iteration*. To ease reading, this dissertation favors the distinction of "intra-iteration" vs "loop-carried."

carried *with respect to loop L iff there is a dynamic instance t_i of t which executes during the i-th iteration of L and a dynamic instance u_j of u which executes during the j-th iteration of L with $i \neq j$ such that there is a dependence from t_i to u_j.*

Similarly, a dependence from t to u is intra-iteration *with respect to L iff there are dynamic instances t_i, u_i of t, u, respectively, which both execute during the i-th iteration of L such that there is a dependence from t_i to u_i.*

Note that the loop-carried designation at loop L has no relation to the loop-carried designation at parent loops of L nor at child loops of L. When context implies a unique loop, this dissertation simply uses "loop-carried dependence" and "intra-iteration dependence" without specifying the loop.

2.4 Dependence Graphs

A program dependence graph (PDG) [21, 48] of a program scope identifies each static instruction from that scope with a vertex, and identifies each control and data dependence among those instructions with a directed edge. Program dependence graphs provide a convenient representation for program transformations.

The PDGs employed in this dissertation differ from Ferrante et al. [21] in two ways. First, Ferrante et al.'s formulation includes *region nodes* to summarize the common control dependences among control-equivalent blocks. Instead, we follow Ottoni [62] by including only regular nodes in the PDG, drawing control dependence directly to other instruction vertices. Second, dependence edges in this dissertation are annotated as loop-carried or intra-iteration (see Section 2.3), and memory dependences are annotated as flow, anti, and/or output dependences.

2.4.1 Equivalence, Communication, and Synchronization

Horwitz et al. prove that if two programs have isomorphic PDGs, the programs are strongly equivalent [34]. This adequacy result suggests a simple transformation correctness criterion: a transformation must "respect" every dependence to generate an isomorphic PDG. One argues the correctness of an automatic thread extraction system by arguing how it respects each dependence despite threaded execution. Scheduling dependent operations to the same thread of execution in the same relative control-flow position naturally preserves the dependence among them. However, when dependent operations are assigned to different threads, the compiler must insert additional communication or synchronization operations to simulate the dependence across thread boundaries. Communication primitives simulate register data dependences by carrying values that would otherwise flow through a register. Synchronization primitives delay memory accesses to prevent data races corresponding to violated memory data dependences.

At an architectural level, communication and synchronization are generally costly operations. Commodity x86 multicore systems provide no specialized core-to-core communication channels; thus, inter-thread communication passes through the memory hierarchy and competes for memory bandwidth with the rest of the application. Synchronization is inherently costly, since it forces some threads to stall thereby reducing utilization.

Communication and synchronization latencies are necessary for correctness, and cannot be eliminated. However, choice of parallel schedule determines whether those latencies penalize the application's critical path. When possible, a parallelization system should choose a parallel schedule in which no communication or synchronization is necessary between threads. Unfortunately, such *embarrassingly parallel* applications are rare in practice.

2.4.2 Pipeline Execution, Dependence Cycles, and the $\mathrm{DAG}_{\mathrm{SCC}}$

The *pipeline execution model* allows a restricted case of inter-thread communication and synchronization which hides those latencies in the steady state, instead paying these la-

tencies once during *pipeline fill*. The key to pipeline execution is acyclic communication and synchronization. This communication pattern allows a *pipeline stage* to perform useful work while subsequent stages stall for communication or synchronization.

To achieve pipeline execution, a thread extraction system partitions operations into ordered pipeline stages such that dependences follow pipeline order. More formally a pipeline partition features acyclic communication if there is an ordering of the stages s_i such that whenever there is a dependence from operation $t \in s_i$ to operation $u \in s_j$, we have $i \leq j$.

Note that, in the absence of dependence cycles, a topological sort of the PDG constitutes a valid pipeline partition. Techniques such as DSWP [63] and PS-DSWP [71] extract threads from general PDGs by identifying dependence cycles as the unit of scheduling. To find these cycles, the DSWP-family techniques compute the Strongly-Connected Components (SCCs) of the PDG and condense the components to vertices. The resulting graph is called the $\mathrm{DAG}_{\mathrm{SCC}}$ or the condensation of the PDG. Components from the $\mathrm{DAG}_{\mathrm{SCC}}$ are scheduled across pipeline stages to minimize imbalance. Figure 2.2 illustrates this process, showing a CFG, PDG, $\mathrm{DAG}_{\mathrm{SCC}}$, and finally parallel execution.

2.5 Dependence Analysis

Dependence analysis algorithms disprove dependence between a pair of operations or conservatively report that those operations *may depend*. Algorithms to accomplish this goal generally focus on one of the conditions of data dependence (Definition 3): disproving *aliasing*, disproving a *feasible path*, or proving a *killing operation* exists along all feasible paths. Each of these conditions has been studied independently.Many algorithms exist for alias (or points-to) analysis [4, 11, 55, 80, 52, 91]. The related problem of shape analysis computes a description of the connectedness or cyclicity of linked data structures [26, 28, 77]; interpreting these shape descriptions answers some alias or points-to queries. Array dependence analysis algorithms focus on the restricted case of pointers

whose values evolve as affine functions of loop induction variables [7, 67]. Several invocations of the must-alias judgment determine whether a `store` operation kills a flow.

In general, dependence analysis is undecidable [49]. Practically, this means that no analysis algorithm precisely determines the presence or absence of dependences on every input. Instead, algorithms deliver precise results only for restricted classes of inputs—their *region of precision*—and deliver conservative results otherwise. Different analysis algorithms potentially feature different regions of precision.

Different analysis algorithms deliver different regions of precision for different costs [32]. Many have observed that more precise analysis algorithms tend to run more slowly than less precise algorithms [22, 32] or that precise algorithms must scale poorly [33, 60], thus discouraging the use of precise yet slow analysis algorithms. On the other hand, high-precision analysis algorithms enable aggressive optimization [15, 27, 71, 74, 88]. This creates a design trade-off: a slow compiler generating fast code, or a fast compiler generating slow code.

2.5.1 Demand-driven: Algorithm and Interface

Classical "all-at-once" analysis algorithms are devised as a minimum fixed-point computation over a set of simultaneous equations [5]. Such algorithms iteratively refine a value for all statements in the program until the convergence. Consequently, determining this value for any statement is as expensive as determining the value for all statements.

Precise analysis can be slow. To reduce compilation times, some authors observe that optimization relies upon only a fraction of analysis facts. Consequently, computation spent on other analysis facts constitutes wasted effort. Those authors propose demand-driven analysis algorithms which expend effort only as necessary to answer specific analysis queries, thus saving time by not considering the rest of the program [29, 79, 96]. Indeed, Chapters 4 and 6 demonstrate than only about half of all memory dependences are necessary for pipeline parallelization.

This thesis relies on a demand-driven *interface* to analysis algorithms (see Chapter 3), but does not require that analysis implementations be demand-driven. Several implementations are wholly all-at-once, several are wholly demand-driven, yet the balance are a hybrid (see Appendix A).

2.6 Speculative Dependence Identification

A thread extraction system must view the PUO through the lens of dependence identification. Fundamental limitations of static dependence analysis mean that the view will conservatively approximate the PUO's ideal dependence structure. Speculative dependence identification allows an optimization system to refine that approximation to better reflect the expected-case program behavior.

Figure 2.2 illustrates the parallelization of a simple loop. The top half demonstrates non-speculative parallelization. In contrast, the bottom half employs speculative dependence identification to extract a more efficient parallel schedule from the same loop.

Central to speculative dependence identification is the notion of a simplifying assumption of program behavior. A speculative assumption is a true or almost-always true property of the PUO which is difficult or impossible to determine through static analysis.

A speculative optimization system may assume various properties of the PUO. For example, if there is reason to believe that a certain conditional branch in the PUO is heavily biased, the optimization system may speculate that the branch transitions unconditionally to its more likely destination (bottom of Figure 2.2). Under this assumption, the conditional branch `if(rare)` does not source any control dependences (because it is speculatively unconditional), and the array update `array[*]=0` does not source any memory dependences (because it is speculatively unreachable). To validate this assumption at execution time, the speculative transformation inserts code to signal misspeculation at the less likely destination of the speculated branch.

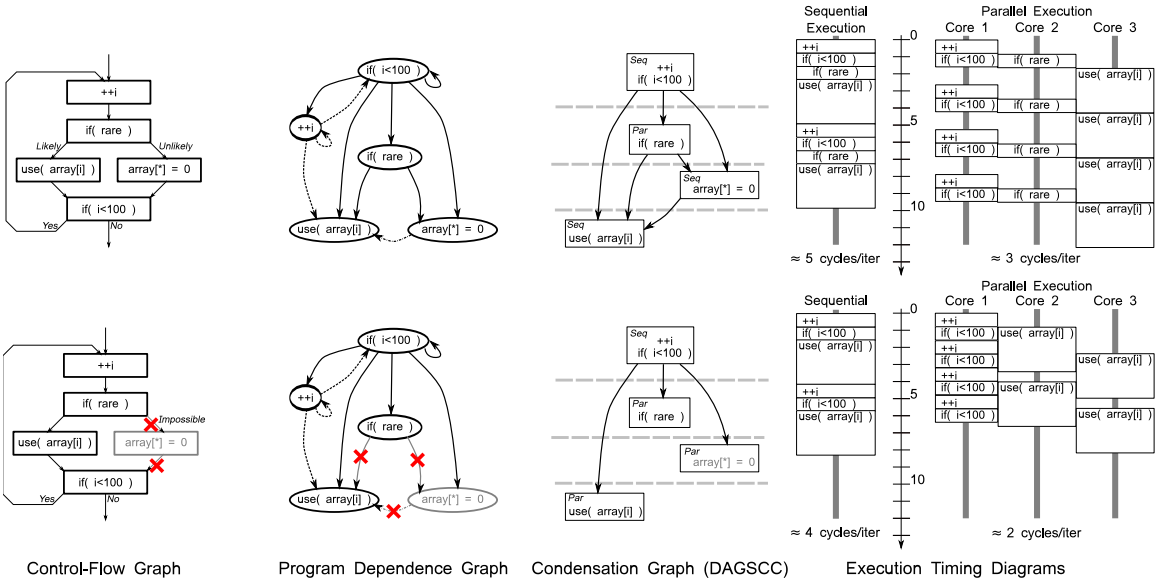


Figure 2.2: Speculative assumptions remove several dependences from a program's dependence graph. (*above, left to right*) The CFG of a loop body either *uses* or *updates* an array. The PDG includes a loop-carried memory dependence edge, ordering updates before subsequent uses, even though updates are unlikely. In the $\mathrm{DAG}_{\mathrm{SCC}}$, the uses are assigned to a *Sequential* component because of the loop-carried constraints from the update. Parallel execution improves throughput over sequential execution. (*below*) By assuming that the heavily-biased branch is unconditional, certain unlikely operations become speculatively dead and thus cannot source dependences. The elimination of such dependences cascades through the PDG and $\mathrm{DAG}_{\mathrm{SCC}}$, ultimately allowing more efficient sequential and parallel schedules. (*Not shown*) these execution diagrams omit validation overheads.

More broadly, speculation is applied to various classes of assumptions: that certain load instructions always read the same value [40]; that transactions are serializable [45, 46, 68, 84]; that pointers reference a restricted class of objects [40]; that certain load instructions read values defined by a store in the *same* iteration [20, 40, 76, 93]; that certain memory objects are short-lived [41, 45]; or that the elements within a linked data structure change infrequently [72]. None of these assumption classes is perfect or complete; each offers various enabling effects, validation costs, and misspeculation rates.

2.7 A Brief Overview of the LLVM IR

This dissertation builds on the LLVM Compiler Infrastructure [51], which provides front-ends for several popular languages, an intermediate representation (IR), robust implementations of "textbook" optimizing transformations, and back-ends for several common architectures.

The LLVM IR organizes each compilation unit as a *module* which contains global symbols: constant byte sequences, global variables, and functions. *Functions* are organized as control-flow graphs (CFGs) of basic blocks, and each *basic block* contains virtual instructions. These *instructions* are "low-level" in the sense that most are trivially mapped onto few machine instructions. For example, the LLVM IR includes instructions representing `load`, `store`, `add`, and `call` with the usual meanings. The LLVM IR largely follows a `load-store` design. The few exceptions, such as LLVM intrinsic instructions representing atomic compare-and-swap or POSIX `memcpy`, can be treated, conservatively, as procedure calls with known, axiomatized behavior.

Instructions in the LLVM IR are normalized to Static Single Assignment (SSA) form [18]; every named value in the program has precisely one static definition. We will often use a static instruction's name to specify all dynamic values computed by that instruction; context should indicate whether we are referring to the instruction or to values it computes.

Values within the LLVM IR are typed, and operations must be compatible with types of their operands. However, the LLVM IR is not strongly typed: the `bitcast` instruction changes a value's type thereby allowing operations inconsistent with the value's declared type. Still, these types have merit. LLVM type checking facilitates compiler development by detecting a broad class of invalid code generated by buggy transformations. Also, LLVM types abstract architectural details such as address alignment constraints and aggregate layout constraints. In particular, LLVM features type-safe pointer arithmetic via the `getelementptr` instruction. Rather than manipulating pointers with integer arithmetic, `getelementptr` offsets a base pointer with integer indices scaled by target-specific aggregate layout constants.

The LLVM memory model is not fully specified, but a partial specification can be inferred from "reasonable" transformations performed by stock optimizations. The Vellvm project observes [95] that the LLVM memory model is consistent with the CompCert memory model. CompCert models each distinct allocation unit as a contiguous *block* of bytes which is separate from all other blocks; storage locations within each block retain their last-stored value during their lifetime, and block lifetimes are delineated by allocation and deallocation routines [54]. These properties are consistent with the few guarantees provided by the C11 memory model, as outlined in Sections 6.2.4, 6.5.6, and 7.22.3 of ISO/IEC 9899-201x [1]. When necessary, this dissertation defers to the CompCert memory model to compensate for underspecified behavior in LLVM.

Chapter 3

The Collaborative Dependence Analysis Framework

"Gettin' good players is easy.
Gettin' 'em to play together is the hard part."
—Casey Stengel

Precise dependence identification enables greater transformation freedom [15, 27, 71, 74, 88]. However, dependence analysis is undecidable [49], and various simplifications of the problem are intractable [33, 60]. Speculation enables transformation despite limitations of dependence analysis [76, 20, 58, 68, 81, 46, 45, 40, 93] yet suffers from validation overheads. These validation overheads may exceed hardware capacity and negate performance improvements [13, 45]. Improving the precision of dependence analysis reduces validation overheads or obviates the need for speculation entirely. This dissertation posits that both speculation and strong dependence analysis are necessary for modern, aggressive optimizations such as thread extraction.

This chapter presents the Collaborative Dependence Analysis Framework (CAF). CAF allows a compiler developer to improve the precision of dependence analysis gradually by adding simple and modular analysis implementations to an ensemble. Beyond simple

composability, CAF offers a design pattern wherein disparate modular analysis algorithms combine their strengths to solve mixed-logic queries (see Figure 3.3). Consequently, each additional dependence analysis algorithm yields a multiplicative improvement to overall ensemble precision, rather than additive improvements from simple composability.

We call this type of composition *collaboration*. To achieve collaboration, each analysis algorithm is structured in a non-conventional manner. Using an understanding of each algorithm's *partiality* (Section 3.1.3) and *decomposition* (Section 3.1.4), these algorithms isolate *foreign premises*—facts about the program which the analysis algorithm needs in order to make further derivations, yet which cannot be derived by this algorithm alone. A collaborative analysis formulates foreign premises in the native query language. The ensemble delegates those *foreign premise queries* to other analysis algorithms to combine the strengths of its members.

Collaboration is a powerful tool for compiler developers who wish to achieve dependence analysis precision. Instead of pursuing precision through increasingly complicated analysis algorithms, collaboration allows the developer to decompose the problem of analysis into many simple implementations which are developed independently. Redundancies in the deductive rules of disparate analysis algorithms can be eliminated and instead be serviced via foreign premise queries. Under this development model, the developer seeks diversity among the set of analysis logics in the ensemble. The CAF allows the compiler developer to achieve dependence analysis precision through many small and simple analysis algorithms rather than a large and complicated algorithm.

3.1 Background

Analysis algorithms for the dependence relation attempt to disprove one or more of the conditions of dependence (Definition 3). This section presents two examples to better illustrate dependence analysis and collaboration between algorithms. These algorithms are named

Array of Structures and *Unique Access Paths*. Neither algorithm is all-encompassing; in fact, the design of each assumes that each will serve as part of a larger ensemble.

3.1.1 Example: Array of Structures

The *Array of Structures* (AoS) algorithm disproves dependences between memory operations by proving that certain pointers to nested aggregate types cannot alias one another. To service a dependence query, AoS examines the expressions which compute those pointers to determine whether the expressions match a schema. AoS is inapplicable to queries which access pointers that do not match that schema, and cannot give a precise answer in those cases.

Query Schema: suppose that AoS receives a query comparing two operations. Suppose that the first operation accesses a pointer of the form A=`&a[`i_1`][`i_2`]...[`i_n`]` and that the second operation accesses a pointer with similar construction B=`&b[`j_1`][`j_2`]...[`j_m`]`.[1] If $a = b$, and if pointers a and b have the same type, and if there exists a position k such that indices $i_k \neq j_k$, then the pointers cannot alias. In that case, AoS reports no dependence since the first condition of Definition 3 cannot hold. Otherwise, AoS cannot give a precise answer.

AoS includes simple induction variable and arithmetic reasoning to prove $i_k \neq j_k$. When applicable, AoS is simple yet powerful: it disproves aliasing even if indices i_q, j_q at other positions $q \neq k$ are non-affine or otherwise incomparable. However, AoS does not contain any logic to test whether $a = b$. Instead, this will be handled as a *foreign premise query* (described in Section 3.4).

```
1  int a = 0, b = 0;
2  int *p = &b, *q = &b;
3  if( -- )
4    p = &a;
5
6  // store 1
7  a = 1;
8
9  // load 1
10 use( *p );
11
12 // load 2
13 use( *q );
```

Points-to sets:
$P(\&\texttt{p}) = \{\&\texttt{b}, \&\texttt{a}\}$
$P(\&\texttt{q}) = \{\&\texttt{b}\}$

Query: *does* `load 2` *depend on* `store 1`*?*
...only if $\&a$ may-alias q [by Def 3]...
...only if $\&a$ may-alias $\&b$ [by UAP]...
Foreign premise query: *may* $\&a$ *alias* $\&b$*?* $\Rightarrow$ No.
Response: No, there can be no dependence.

Query: *does* `load 1` *depend on* `store 1`*?*
...only if $\&a$ may-alias p [by Def 3]...
...only if ($\&a$ may-alias $\&b$ or $\&a$ may-alias $\&a$) [by UAP]...
Foreign premise query: *may* $\&a$ *alias* $\&b$*?* $\Rightarrow$ No.
Foreign premise query: *may* $\&a$ *alias* $\&a$*?* $\Rightarrow$ Yes.
Response: Unknown, there may be a dependence.

Figure 3.1: *(above left)* Locations `&a`, `&b` are captured on lines 2 and 4. Locations `&p`, `&q` are never captured throughout the program. *(below left)* UAP accumulates a points-to set of values stored into each of the non-captured locations. *(above right)* UAP decomposes the query into a simpler premise and issues a foreign premise query. An unspecified other analysis algorithm disproves the foreign premise query. UAP reports a result which relies on three logics: Def 3, UAP, and something else. *(below right)* UAP decomposes the query into simpler premises and issues foreign premise queries. No other analysis algorithm disproves the foreign premise queries. UAP reports *unknown*.

3.1.2 Example: Unique Access Paths

The *Unique Access Paths* (UAP) algorithm disproves dependences between memory operations by disproving a certain class of aliasing between pointers loaded from memory. At initialization, it performs a linear scan over the entire module to collect *points-to* sets for certain `load` instructions. It then services dependence queries which match its schema. UAP is inapplicable to queries that do not match the query schema.

Initialization scan: by scanning the entire module, UAP identifies a subset of global, stack, or heap storage locations whose *addresses* are never *captured*, i.e., never stored into memory and never passed to an externally defined function (see Figure 3.1). Without captures, pointers to such storage locations can only propagate via virtual registers—never through memory. Consequently, the compiler may easily enumerate every load from or store to *non-captured* storage locations by tracing register data flow. This is a significant simplification over the general case, in which pointers to those storage locations may propagate through memory.

Next, UAP considers every `store` instruction which writes a value into a non-captured storage location. UAP accumulates the set $P(L)$ of values stored into each non-captured storage location L (see the bottom left of Figure 3.1).

Query Schema: suppose that UAP receives a query comparing two operations. Suppose that the first operation accesses a pointer A and the second accesses a pointer B. Further, suppose that pointer A is computed as A=`load` L, where L is a pointer to a storage location which the initialization scan identifies as non-captured. UAP reasons that A may-alias B only if there is some $p \in P(L)$ such that p may-alias B. Further, if p is the unique member of $P(L)$, and if p must-alias B, then A must-alias B.

For example, when considering Figure 3.1, `load 1` dereferences the pointer `p`. The storage location at `&p` is non-captured, and the initialization-scan determines that two val-

[1]The bracket notation represents adding the pointer expression on its left to a multiple of the quantity within the brackets; it does *not* access memory. It corresponds to LLVM's `getelementptr` instruction (see Section 2.7). "Aggregate" refers to C's array and structure types, but not union types.

ues are stored into `&p`: the pointer `&b` on line 2, and the pointer `&a` on line 4. Thus, the expression `*p` dereferences either the pointer `&a` or the pointer `&b`. A query comparing `load 1` with `store 1` is rewritten as two premises, one comparing `&b` with `&a`, and the other comparing `&a` with `&a`. These premises are easier to resolve than the original query.

When applicable, UAP is powerful since it provides a means to trace the flow of pointers through simple data structures. UAP contains logic that equates a value loaded from a storage location with the set of values stored into that storage location. However, UAP does not contain any logic to test whether p may-/must-alias with B for all $p \in P(L)$. Instead, this is handled as a *foreign premise query* (described in Section 3.4).

3.1.3 Partiality and Algorithmic Diversity

Many logics have been applied to the related problems of alias analysis and dependence analysis. Each implementation has limitations, but a diversity of logics will often hide those limitations.

Since dependence analysis is vital for a myriad of compiler optimizations and other applications, the literature is rich with algorithms that compute conservative approximations of dependence analysis. Each algorithm is designed to give satisfactory results for certain restricted cases of the greater problem [31]. For example, research projects which study the optimization of programs which regularly traverse multidimensional arrays discover dependence analysis algorithms which precisely analyze linear integer arithmetic (LIA), yet which are imprecise for other inputs [67]. In general, we say that each analysis logic has a *region of precision* (RoP), that is, a subset of all dependence queries for which the logic derives a precise dependence result.

The RoP describes the limitations of a single dependence analysis algorithm. However, RoPs help us understand the relative strengths of algorithms in an ensemble. Consider three hypothetical analysis algorithms with different RoPs. Figure 3.2 presents a visual analogy. In the example, the first analysis algorithm has a smaller RoP than the third algorithm

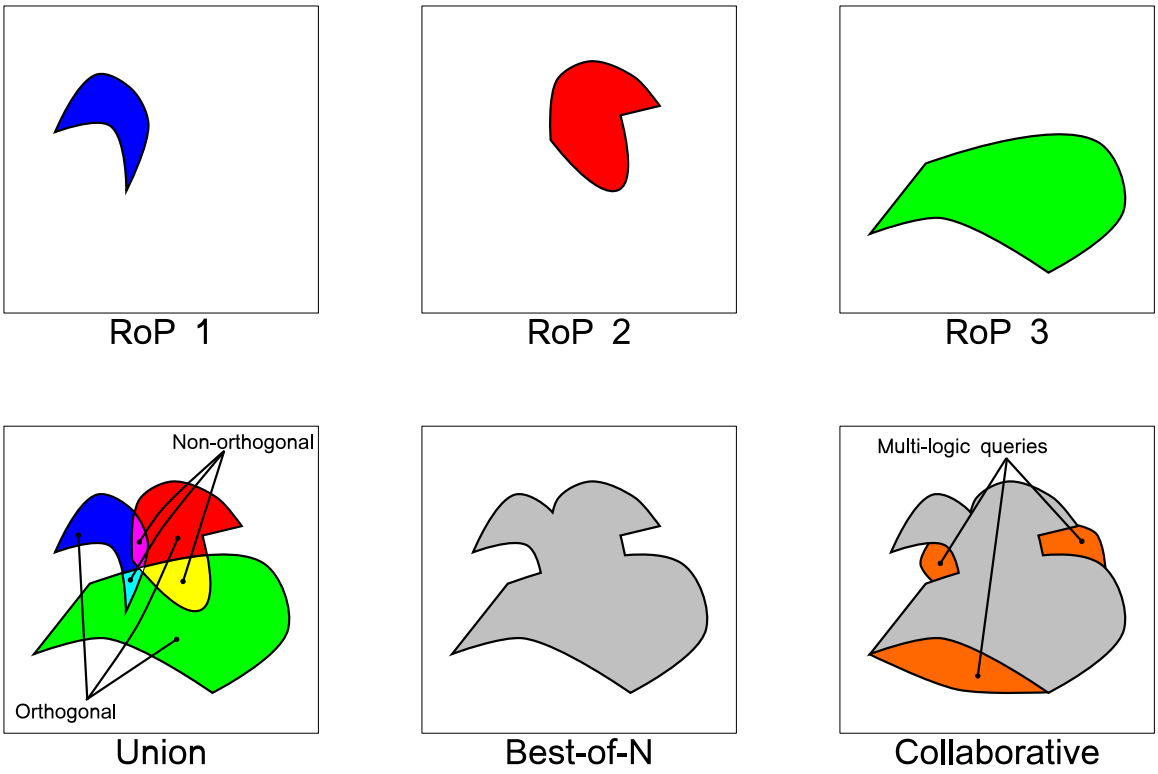


Figure 3.2: Regions of Precision in the space of all dependence queries. (*RoP 1, 2 and 3*) Three hypothetical logics precisely recognize only a subset of all dependences, termed their *regions-of-precision* (RoP). (*Union*) The three RoPs may partially overlap, in which case we say that the logics are non-orthogonal. (*Best-of-N*) A *best-of-N* ensemble logic chooses the most precise dependence result from any member logic. The RoP for a best-of-N ensemble is the union of the RoPs of its member logics. (*Collaborative*) A *collaborative* ensemble logic additionally recognizes queries which no member analysis recognizes alone.

(compare sizes of RoP 1 and RoP 3). Superficially, one may claim that the third algorithm is more precise than the first. However, the union of these three RoPs demonstrates that algorithm 1 disproves certain queries which algorithm 3 cannot. Thus, there is no strict dominance relationship among these algorithms.

In general, a demand-driven dependence analysis algorithm receives queries and responds with an answer indicating the *definite absence* or the *possible presence* of a memory dependence. "Possible presence" is the conservative, external representation of two cases: definite presence of a dependence, or unknown. Internally, certain analysis logics can distinguish these cases. Indeed, this distinction enables best-of-N collaboration.

Example algorithms AoS (Section 3.1.1) and UAP (Section 3.1.2) illustrate partiality concretely. Both logics report unknown if the query does not match its respective query schema. AoS also reports unknown if it cannot prove a pair of indices unequal, and UAP if it cannot compare the loaded pointers. These unknown cases are fundamentally different than proving the existence of a dependence.

3.1.4 Decomposition and Multi-Logic Queries

Some analysis logics "decompose" an input query into one or more premises. The example AoS may decompose a query between large pointer expressions A and B into a premise concerning the simpler pointer expressions a and b (Section 3.1.1). Similarly, the example UAP may decompose a query between a pointer A=`load` L and a pointer B into several premises concerning each value p loaded by A and the pointer B. Such decomposition is fundamental to proving any statement and is not unique to CAF.

CAF uses decomposition as a means of structuring analysis algorithms and as a criterion for modularization. Beyond simple decomposition, CAF encourages analysis algorithms to isolate these premises as foreign premise queries (Section 3.4) and thus achieve collaboration (Section 3.4). Critically, this allows the CAF to disprove multi-logic queries.

Figure 3.3 illustrates a proof with statements as circles and derivational rules as arrows.

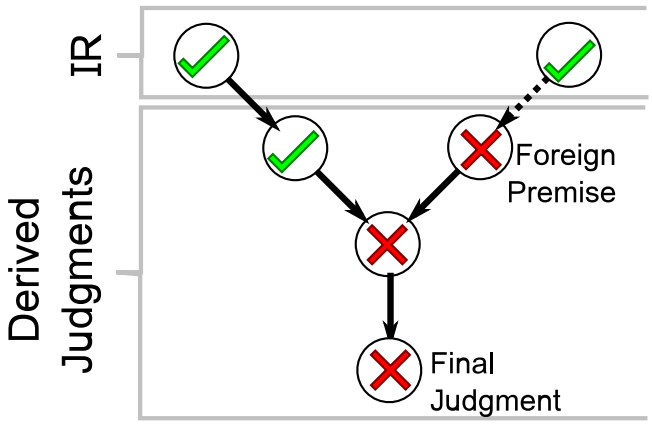


Figure 3.3: Why do multi-logic queries exist? Circles represent various statements about the program, ranging from facts directly observed in the IR to the final dependence judgment. Analysis logics (arrows) allow us to derive new statements from established premises. Two hypothetical dependence logics are shown here, corresponding to the solid and dashed arrows. The solid-arrow logic derives three statements, yet cannot derive the final statement since the *foreign premise* cannot be established using only solid arrows. The dashed-arrow logic establishes the *foreign premise*, yet cannot establish other statements. By combining both, a collaborative ensemble establishes the final statement.

The top-most statements codify facts directly observable in the IR, for instance, the concrete sequence of static instructions that compute a pointer. The bottom-most statement asserts the independence of two operations, i.e., that there is no feasible execution which satisfies the definition of a memory dependence. The final judgment is derived from the IR through a series of derivational steps, where each derivational step is drawn from an analysis logic.

In the multi-logic query scenario, an analysis logic derives some statements yet is unable to derive all statements. For instance, if an analysis logic provides the solid arrows in Figure 3.3, it can only derive those statements marked with a check; the final judgment cannot be derived. Multi-logic queries arise as a consequence of undecidability; any analysis algorithm—no matter how elaborate—fails to prove some true statements.

By adding another algorithm which provides the dashed arrows in Figure 3.3, the final judgment is established through the collaboration of two algorithms. Collaboration allows an ensemble to disprove multi-logic queries, where simple best-of-N composition fails.

3.1.5 Combining Analysis Implementations

Given a diversity of analysis logics, there are several ways to combine them into an ensemble. This section briefly reviews some other methods.

Nelson and Oppen present an algorithm to combine decision procedures for several logics into single decision procedure for the combined logic [61]. Their method separates a multi-logic expression into several single-logic expressions and links each to a boolean satisfiability query using the equality operator common to all logics. A decision procedure for boolean satisfiability processes these constraints in tandem with decision procedures for each logic.

Click and Cooper [16] study combinations of monotone analysis frameworks. Given two abstract domains, they solve both problems simultaneously as usual, but additionally introduce *interaction* functions which carry knowledge across domains. The tandem algorithm still computes the greatest fixed point in each domain provided that the interaction functions are monotonic. However, the tandem analysis only improves precision when the analysis developer creates interaction functions. This technique requires the analysis developer to devise interaction functions for each pair of abstract domains, resulting in manual work quadratic in the number of analysis implementations. In contrast, the CAF allows the analysis developer to specify how each analysis instigates interaction with other analysis implementations, hence manual work proportional to the number of implementations.

Lerner et al. [53] present an algorithm to combine monotone analysis frameworks while reducing the manual costs. Observing that compiler developers often introduce analysis implementations to support particular code transformations, Lerner et al.'s method repurposes the code transformation as an implicit communication among analysis algorithms. Specifically, as analysis algorithms discover new facts about the input code, they replace regions of the IR with a simpler, equivalent IR fragment. Other analysis implementations recompute their results for simplified regions of code, thus incorporating facts from peer analysis algorithms into their own conclusions.

IR replacement obviates the need for manual specification of interaction functions prescribed by Click and Cooper [16]. Further, Lerner et al. argue that IR replacement requires less manual effort than interaction functions because the compiler developer must already implement code transformation corresponding to the analysis. This presupposes that each analysis implementation is strongly-coupled with an optimizing transformation. Further, it assumes that these optimizing transformations produce an equivalent IR which is simpler or otherwise more amenable to analysis. This assumption is not necessarily true for heroic transformations, such as automatic thread extraction. For instance, thread extraction may replace simple register data-flow with inter-process communication, which is generally *more difficult* to analyze. Hoopl [73] provides (among other things) a concrete implementation of Lerner et al.'s approach in Haskell.

The CAF combines arbitrary analysis algorithms, not simply decision procedures or monotone frameworks. The CAF specifies interaction among the implementations without coupling analysis to transformation and without overburdening the developer with quadratically-many interaction functions.

3.2 Structure of an Analysis Implementation in Isolation

A traditional dependence analysis algorithm can be viewed as a black box: queries come in, responses go out, as in the leftmost diagram of Figure 3.4. The CAF refines this view in two critical ways which correspond to the notions of partiality (Section 3.1.3) and decomposition (Section 3.1.4).

Partiality dictates that any analysis algorithm gives precise results for queries within its RoP and imprecise results for queries outside its RoP. Acknowledging partiality as fundamental, the CAF provides a means for analysis algorithms to distinguish precise results. The central diagram in Figure 3.4 refines the traditional by adding a "chain" port; whenever an analysis algorithm is unable to give a precise result, it instead passes the unmodified

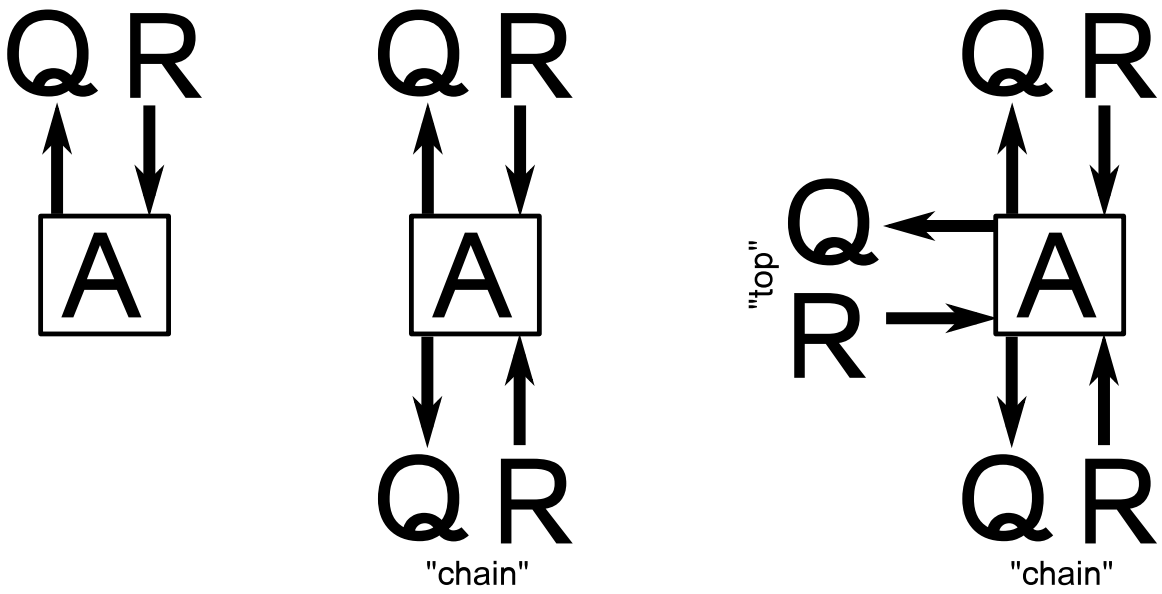


Figure 3.4: *(left to right)* Traditional dependence analysis algorithms accept queries Q and return responses R, yet do not distinguish precise and imprecise answers. A "chain" port allows an algorithm to declare that it cannot give a precise answer for the given query, and instead delegate it to some other analysis algorithm. Stringing such analyses end-to-end achieves best-of-N composition. A "top" port provides an path for foreign premise queries.

query to its chain port, and reports whatever response it receives from its chain port.

Decomposition into foreign premise queries means that the ultimate derivation may branch into several simpler derivations. The rightmost diagram in Figure 3.4 refines the central diagram by adding a "top" port. As the analysis algorithm generates foreign premise queries, it passes these along its top port, and incorporates the corresponding responses into its ultimate derivation.

The distinction between the chain and top ports is important. The chain port ensures that every analysis algorithm, in turn, has a chance to consider any unsolved queries, and implicitly indicates that earlier analysis algorithms on the chain cannot determine a precise result. The top port, however, indicates that the query is new and that there is no indication of which analysis algorithms may solve it. This distinction will become more clear in Section 3.4 and evaluated in Section 6.1.2.

More concretely, Figure 3.5 shows the example AoS (Section 3.1.1) in terms of the three-port interface from Figure 3.4. Each analysis algorithm is implemented as a separate

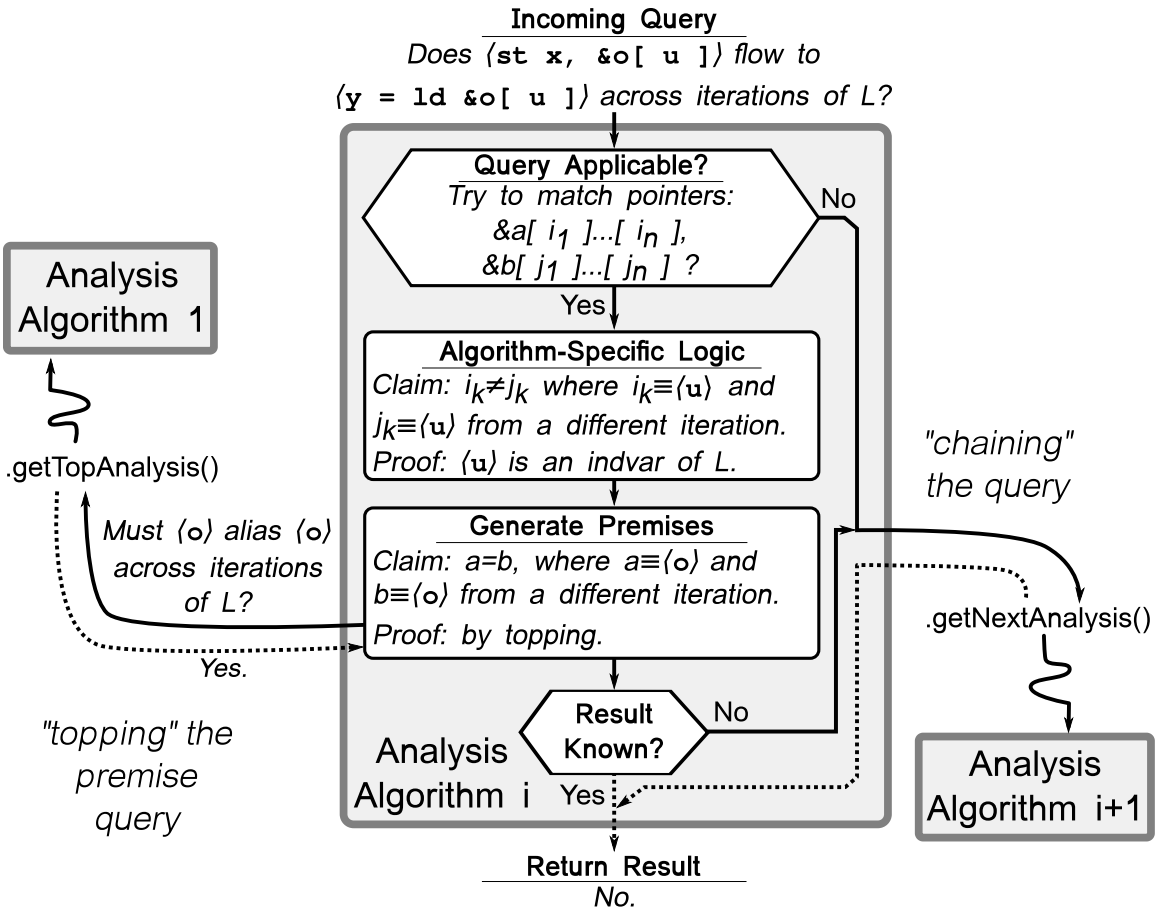


Figure 3.5: Flowchart depicting the internal operation of a generic analysis algorithm, showing a specific trace of the example AoS (Section 3.1.1).

class featuring methods corresponding to each type of query (see Section 3.3). To pass a query along the chain port, an analysis implementation retrieves a reference to the next analysis and invokes the appropriate query method on it. Similarly, to pass a query along the top port, an analysis implementation retrieves a reference to the top analysis and invokes the appropriate query method on it.

3.3 Informal Semantics of the Query Language

Each analysis implementation accepts queries and possibly generates new queries. To achieve composability among implementations despite their various internal models, the collaborative analysis framework defines a simple language of queries and requires that

each implementation accept queries of that form. This section informally describes the query language employed by CAF. The formal semantics of this query language are described in Section 3.7.

Figure 3.1 summarizes the three types of analysis queries used in CAF. These query types correspond to methods of the analysis implementations. `modref_*` queries determine whether a contextualized operation modifies (`Mod`) or reads (`Ref`) some set of resources, returning `None`, `Mod`, `Ref`, or `ModRef` (modifies *and* reads). A *target* parameter implicitly specifies a set of target resources. In `modref_ii`, the set of resources is the resource footprint of a target *operation* i_2 whereas in `modref_ip` the set of resources is the set of memory locations referenced by a s_2-byte access to target pointer *pointer* p_2.

The mod-ref relation is similar to the may-depend relation, as illustrated in Figure 3.6. One determines if there is a memory dependence from operation i_1 to i_2 within a single iteration of loop L, by issuing two queries: `modref_ii`$(i_1,$ `Same`$, i_2, L)$ and `modref_ii`$(i_2,$ `Same`$, i_1, L)$. Similarly, one determines if there is a memory dependence from operation i_1 to i_2 across iterations of loop L, by issuing the queries: `modref_ii`$(i_1,$ `Before`$, i_2, L)$ and `modref_ii`$(i_2,$ `After`$, i_1, L)$. The results of both queries determine if there is a flow dependence, an anti dependence, and an output dependence. A pair of operations may have more than one type of dependence because some operations both read and write memory, for instance, a call site.

3.4 Foreign Premise Queries, Topping, and Ensembles

Some analysis logics decompose a query into one or more premises. If it can establish all of those premises, then the analysis logic is able to derive new facts about the program under analysis. The compiler developer then chooses *how and where* to establish these premises—either as additional derivation rules within the same analysis implementation, or as foreign premise queries delegated to other analysis algorithms.

Semantics	Query	Context and Path Qualifiers
*May instruction i_1 write to (*Mod*) or read from (*Ref*) the resource footprint of target operation i_2?* Return None, Mod, Ref, or ModRef.	modref_ii$(i_1, \texttt{Before}, i_2, L)$	i_1 executes in iteration τ_1 of loop L and i_2 executes in some later iteration $\tau_2 > \tau_1$.
	modref_ii$(i_1, \texttt{Same}, i_2, L)$	i_1 and i_2 both execute in the same iteration of L.
	modref_ii$(i_1, \texttt{After}, i_2, L)$	i_1 executes in iteration τ_1 of loop L and i_2 executes in some earlier iteration $\tau_2 < \tau_1$.
*May instruction i_1 write to (*Mod*) or read from (*Ref*) the resource footprint of target pointer p_2 of length s_2?* Return None, Mod, Ref, or ModRef.	modref_ip$(i_1, \texttt{Before}, p_2, s_2, L)$	i_1 executes in iteration τ_1 of loop L and values of p_2 are computed in later iterations $\tau_2 > \tau_1$.
	modref_ip$(i_1, \texttt{Same}, p_2, s_2, L)$	i_1 executes in the same iteration as p_2.
	modref_ip$(i_1, \texttt{After}, p_2, s_2, L)$	i_1 executes in iteration τ_1 of loop L and p_2 computes pointers in earlier iterations $\tau_2 < \tau_1$.
May any memory location which is referenced by a dynamic pointer value computed by operation p_1 of length s_1 alias with the resources referenced by target pointer p_2 of length s_2? Return No Alias, May Alias, or Must Alias.	alias_pp$(p_1, s_1, \texttt{Before}, p_2, s_2, L)$	p_1 is computed in iteration τ_1 of loop L and p_2 is computed in some later iteration $\tau_2 > \tau_1$.
	alias_pp$(p_1, s_1, \texttt{Same}, p_2, s_2, L)$	p_1 and p_2 are computed during the same iteration of L.
	alias_pp$(p_1, s_1, \texttt{After}, p_2, s_2, L)$	p_1 is computed in iteration τ_1 of loop L and p_2 is computed in some earlier iteration $\tau_2 < \tau_1$.

Table 3.1: Types of queries: modref_ii compares the footprint of the first instruction to the footprint of the target instruction; modref_ip compares instead to the resources referenced by a target pointer; alias_pp compares the resources referenced by two pointers.

Intra-iteration queries

$\texttt{modref\_ii}(i_1, \texttt{Same}, i_2, L)$	$\texttt{modref\_ii}(i_2, \texttt{Same}, i_1, L)$ None	Mod	Ref	Mod-Ref
None	–	–	–	–
Mod	–	O	T	T, O
Ref	–	A	F	F, A
Mod-Ref	–	A, O	T, F	T, F, A, O

Loop-carried queries

$\texttt{modref\_ii}(i_1, \texttt{Before}, i_2, L)$	$\texttt{modref\_ii}(i_2, \texttt{After}, i_1, L)$ None	Mod	Ref	Mod-Ref
None	–	–	–	–
Mod	–	O	T	T, O
Ref	–	A	F	F, A
Mod-Ref	–	A, O	T, F	T, F, A, O

Figure 3.6: Using two `modref_ii` queries to decide a dependence between operations i_1 and i_2. (*left*) **Intra**-iteration queries use the temporal relation `Same`. (*right*) Loop-carried (or **inter**-iteration) queries use temporal relations `Before` and `After`, respectively. The keys T, F, A, O correspond to true (read-after-write), false (read-after-read), anti (write-after-read), and output (write-after-write) dependences.

A broad class of premises can be represented as foreign premise queries. Applicability is limited to the types of queries listed in Section 3.3; it is not always possible to formulate the premise as a foreign premise query. However, when possible, this dissertation encourages the latter approach, since it simplifies ensemble development and contributes to collaborative performance. Under the prescribed development strategy, each implementation should consider "topping" a foreign premise query as a call into an oracle. Even if the oracle is imperfect, it improves with time, and topping analysis implementations receive these improvements automatically.

Topping delegates a query to the entire strength of the ensemble. This is achieved by forming an ensemble: each algorithm is composed such that one algorithm's chained queries feed the next. The chain ends with a null algorithm which always reports the most conservative answer. Client-generated queries enter the ensemble's top-most member, and visit each analysis in turn until a precise answer is found or the null algorithm admits defeat. If an algorithm generates foreign premise queries, it "tops" those queries by feeding them

into the top-most algorithm (see Figure 3.7).

Without topping, composition by chaining alone resembles the means of composition in LLVM [51]. Topping can significantly improve an ensemble's performance as evaluated in Section 6.1.2.

3.4.1 Example: Solving a Mixed-Logic Query with AoS and UAP

To illustrate collaboration, this section traces through a multi-logic query that is solved through collaboration of AoS (Section 3.1.1) and UAP (Section 3.1.2).

In the example code from Figure 3.8, an allocation is captured into two pointers `src` and `dst` during program initialization. Elsewhere, the procedure `MacGuffin` traverses `src` linearly, computing some function of its elements, and writes results into `dst`. Although the programmer wrote compound expressions `... = src[i]` and `dst[i] = ...`, the example lists multiple `load` and `store` operations as expanded by the compiler front-end.

Note that this example looks trivial only because it is contrived to seem so. Since the initialization and use of pointers `src` and `dst` are separated by procedural boundaries, interprocedural analysis techniques are required to establish that `p1` and `p2` are equivalent during the first iteration of loop H. In fact, the compiler cannot even hoist loads `L1` and `L3` out of the loop until it establishes that store `S1` never modifies `src` or `dst`, and thus pointers `p1` and `p2` are loop invariant.

One may wish to parallelize the loop in the example. To prove that parallelizing this loop (and potentially re-ordering its iterations) will not change observable program behavior, the compiler must first prove that every iteration is independent of all others. Assuming that the function `f` is pure (i.e., neither accesses memory nor issues side effects), the compiler uses seven queries to establish independence of iterations.

1. $\mathrm{modref\_ii}(\texttt{S1}, \texttt{Before}, \texttt{S1}, \texttt{H})$ tests whether store `S1` writes a value that `S1` will overwrite in later iterations, i.e., whether there is a loop-carried output dependence.

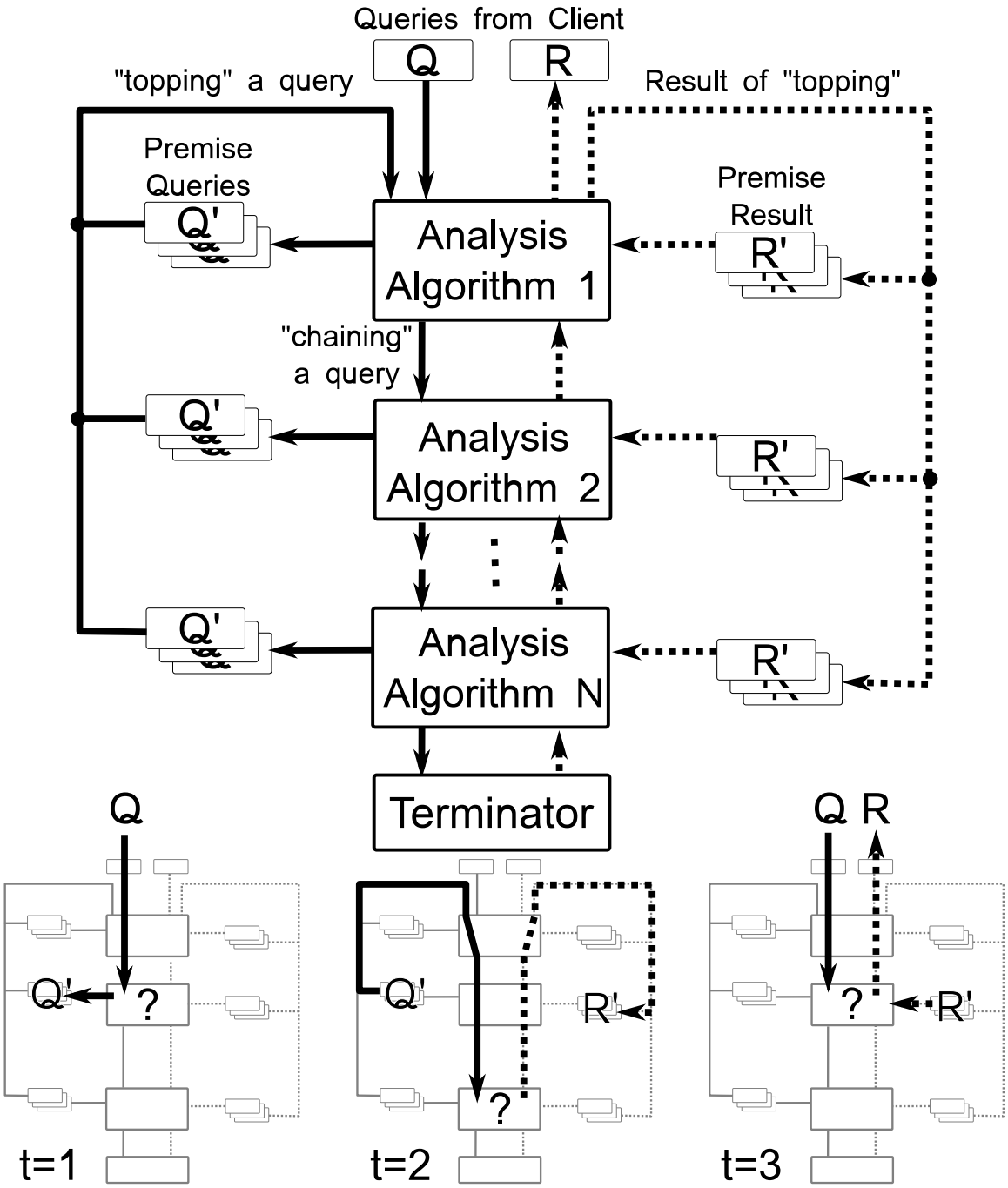


Figure 3.7: *(top)* Several algorithms comprise an ensemble. Premise queries re-enter at the top of the ensemble. Chained queries pass to the next algorithm in the ensemble. The ensemble ends with a terminator which returns the conservative answer. *(bottom)* Path of a query Q over time. $t = 1$: Algorithm 2 deconstructs Q into a foreign premise query Q'. $t = 2$: Algorithm N solves Q' and returns the R'. $t = 3$: Algorithm 2 receives R' and combines it with other information to yield the result R for Q.

```
1  // global scope
2  int myArray[N] = {0};
3  static int *src, *dst;
4
5  int main() {
6    src = myArray;
7    dst = myArray;
8    ...
9    work();
10 }
11 ...
12 void MacGuffin() {
13   for(i=0; i<N; ++i) {          // Loop H
14     // The programmer wrote  '... = src[i];'
15     // which results in two loads:
16     register int *p1 = * &src; // Load L1
17     register int tmp = p1[i];  // Load L2
18
19     // The programmer wrote 'dst[i] = ...'
20     // which results in a load and a store:
21     register int *p2 = * &dst; // Load L3
22     p2[i] = f( tmp );          // Store S1
23   }
24 }
```

Figure 3.8: Example code uses both affine iteration and storage of pointers into memory. Understanding this code requires a theory of linear integer arithmetic and a theory of the reachability.

2–4. For each of `L1`, `L2`, and `L3`: the query $\mathrm{modref\_ii}(\texttt{Li}, \texttt{Before}, \texttt{S1}, \texttt{H})$ tests whether load `Li` reads values that `S1` will overwrite in later iterations, i.e., whether there is a loop-carried anti dependence.

5–7. For each of `L1`, `L2`, and `L3`: the query $\mathrm{modref\_ii}(\texttt{S1}, \texttt{Before}, \texttt{Li}, \texttt{H})$ tests whether store `S1` writes a value that `Li` will read in later iterations, i.e., whether there is a loop-carried flow dependence.

Ideally, each of those queries returns `NoModRef`, thus proving that parallelization is safe. If any query returns otherwise, reordering iterations introduces a race condition. The compiler would either avoid parallelization or introduce synchronization.

Finally, suppose that our collaborative ensemble contains four analysis implementations. In order from top to bottom, the ensemble contains:

- AoS: the example algorithm from Section 3.1.1;
- UAP: the example algorithm from Section 3.1.2;
- Basic: a trivial algorithm codifying the knowledge that a named global variable aliases itself; and,
- Null: the terminator analysis, which always reports the least-precise result.

Discussion will focus on the query $\mathrm{modref\_ii}(\texttt{S1}, \texttt{Before}, \texttt{L2}, \texttt{H})$. Servicing this query proceeds as follows:

1. AoS receives the query $\mathrm{modref\_ii}(\texttt{S1}, \texttt{Before}, \texttt{L2}, \texttt{H})$.
2. AoS determines that `S1` accesses the pointer `&p2[i]` which matches the schema $\texttt{\&}a\texttt{[}i_1\texttt{]}$ where $a \equiv \texttt{p2}$ and $i_1 \equiv \texttt{i}$.
3. AoS determines that `L2` accesses the pointer `&p1[i]` which matches the schema $\texttt{\&}b\texttt{[}j_1\texttt{]}$ where $b \equiv \texttt{p1}$ and $j_1 \equiv \texttt{i}$.

4. AoS establishes that $i_1 \neq j_1$ by observing that `i` is an induction variable of loop H, and that this query refers to dynamic instances of $S1, L2$ from different iterations of H (parameter `Before`).

5. AoS formulates a foreign premise query, and tops it: `alias_pp(p2, Before, p1, H)`.

   (a) AoS receives the query `alias_pp(p2, Before, p1, H)`. This query does not match the AoS query schema. AoS chains the query.

   (b) UAP receives the query `alias_pp(p2, Before, p1, H)`.

   (c) UAP determines that `p2` is computed as `p2 = load &dst`, and that `dst` refers to a non-captured storage location.

   (d) UAP determines that `p1` is computed as `p1 = load &src`, and that `src` refers to a non-captured storage location.

   (e) UAP looks up the points-to sets for `dst` and `src`. $P(\&\texttt{dst}) = \{\texttt{myArray}\}$. $P(\&\texttt{src}) = \{\texttt{myArray}\}$.

   (f) UAP issues a foreign premise query for each pair in $P(\&\texttt{dst}) \times P(\&\texttt{src})$. In this case, it tops `alias_pp(myArray, Before, myArray, H)`.

      i. AoS receives the query `alias_pp(myArray, Before, myArray, H)`. This query does not match the AoS query schema. AoS chains the query.

      ii. UAP receives the query `alias_pp(myArray, Before, myArray, H)`. This query does not match the UAP query schema. AoS chains the query.

      iii. Basic receives the query `alias_pp(myArray, Before, myArray, H)`.

      iv. Basic reports `MustAlias` = `alias_pp(myArray, Before, myArray, H)`.

      v. UAP relays the chained result.

      vi. AoS relays the chained result.

   (g) UAP receives the result: `MustAlias` = `alias_pp(myArray, Before, myArray, H)`.

   (h) UAP reports: `MustAlias` = `alias_pp(p2, Before, p1,` H`)`.

(i) UAP relays the chained result.

(j) AoS relays the chained result.

6. AoS receives the result: $\texttt{MustAlias} = \texttt{alias\_pp}(\texttt{p2}, \texttt{Before}, \texttt{p1}, H)$.

7. AoS reports: $\texttt{NoModRef} = \text{modref\_ii}(\texttt{S1}, \texttt{Before}, \texttt{L2}, \texttt{H})$.

3.5 Scheduling Priority

Queries visit each analysis algorithm in an ensemble until a precise result is found. Several analysis algorithms may be able to solve a query, potentially via different independence arguments. The order of analysis algorithms is unimportant from a correctness or precision perspective. Although reordering analysis algorithms may change *which* algorithm provides the precise result, the precise results must concur assuming that all ensemble members are sound. Said another way, sound analysis algorithms in an ensemble commute.

Although order does not affect correctness, order may have a large impact on performance. When several algorithms can provide a precise result via different independence arguments, it is likely that certain arguments are more efficiently formulated, resulting in a performance discrepancy. Suppose, for instance, that an analysis implementation A spends 1ms per query and decides 90% of all queries. Independently, implementation B spends 100ms per query and decides 10% of all queries. An ensemble of A above B services queries in $90\% \times 1ms + 10\% \times (1ms + 100ms) = 11ms$ on average. On the other hand, an ensemble of B above A services queries in $10\% \times 100ms + 90\% \times (100ms + 1ms) = 100.9ms$ on average.

This effect is particularly pronounced in analysis logics which rely on foreign premise queries. Thus, there is an opportunity to improve query latency in the expected case by controlling the relative position (or, height) of analysis algorithms in the ensemble.

The CAF uses the notion of a *scheduling priority* to order algorithms in an ensemble.

Most analysis algorithms use a default scheduling priority, which means that their relative positions are arbitrary. The developer may assign higher or lower priorities to more or less efficient algorithms to place them higher or lower in the ensemble.

Algorithms with low priority receive queries only after high-priority analysis algorithms have chained the queries. Consequently, the low priority algorithms never consider the queries which are easily and efficiently solved by those algorithms higher in the ensemble. This reduces the computational load on slower algorithms, in essence using the more efficient algorithms as a filter.

3.5.1 Ensuring Termination

Topping a premise query introduces a cycle into the ensemble. Developers must ensure that cycles do not cause infinite recursion. Such assurances resemble familiar termination arguments: define a query-metric with a lower bound and demonstrate that premises have a lower query-metric than the original query. The metric depends on the particular analysis algorithm, though a few examples clarify the approach.

AoS (example from Section 3.1.1) must terminate. Consider this query-metric: indexing operations (LLVM's `getelementptr` instruction) have a metric one greater than their base operand; all other operations have a metric of zero. AoS strips one layer of array indexing before issuing a premise query, thus decreasing the metric toward the lower bound. Infinite topping is impossible.

Several analysis algorithms substitute base pointers for address expressions to generate premise queries. For example, the *underlying objects* of the C expression `c ? &p[i] : &q.f` are `p` and `q`. At the IR level, *use-lists* witness the relationship between compound expressions and their operands. In the absence of Φ-nodes use-lists are an acyclic relation. We construct a metric by treating this relation as a partial order. Tracing a pointer to its underlying objects (while avoiding cycles on Φ-nodes) traverses the partial order monotonically, thus decreasing the metric.

When several algorithms generate premise queries, the termination argument must consider all algorithms to rule-out compositional infinite loops. The single-algorithm termination arguments can be extended to the multiple-algorithm scenario either by using a single metric universally, or by demonstrating that no algorithm's premises increase another algorithm's termination metric. Admittedly, neither of these approaches is particularly composable, yet the latter is easy in practice. Indeed, the latter seems conceptually natural; if you interpret a termination metric as a measure of a query's complexity, it's difficult to imagine two analysis implementations whose models employ opposite notions of complexity.

3.6 Analysis Implementations

The Liberty Research Group has implemented many analysis algorithms in this framework which cover a broad range of reasoning types. Table 3.2 briefly summarizes the analysis implementations. Appendix A describes each in depth.

Several of our analysis implementations reason about the semantics of particular instructions or other IR features (Basic Loop, Auto-Restrict, Φ-Maze, Semi-Local Function). SMT-AA reduces many dependence analysis queries into a SMT theorem to be solved by CVC3 [8]; it uses the theories of Linear Integer Arithmetic (LIA), Non-linear Integer Arithmetic (NIA), and Uninterpreted Functions (UF). Other implementations support restricted cases of LIA more efficiently (Array of Structures, SCEV). Two analyses reason about intermediate operations which *kill* a flow dependence (Kill-Flow, Callsite Depth-Combinator). Quite a few implementations reason about heap reachability, i.e., restricting the classes of objects referenced by a pointer field of another object (Global Malloc, No-Capture Global, No-Capture Source, Disjoint Fields, Sane Type, No-Capture Fields, Acyclic, Unique Access Paths, Field Malloc). These reachability implementations were developed in response to the linked-data structures common in general purpose applications.

Analysis Algorithm	Sensitivity				Demand-driven?	Foreign-premise queries
	Memory-flow	Control-flow	Array/field	Calling-context		
Array of Structures	×	×	✓	×	Fully	1
Auto-restrict	×	×	×	✓	Partially	0
Basic Loop	×	×	✓	×	Fully	Many
Callsite	✓	✓	×	✓	Fully	Many
Global Malloc	✓	×	×	×	Partially	0
Kill Flow	✓	✓	×	×	Fully	Many
No-Capture Global	×	×	×	×	Fully	0
No-Capture Source	×	×	×	×	Fully	0
PHI Maze	×	×	×	×	Fully	0
Semi-Local	×	×	×	×	Partially	Many
Unique Paths	✓	×	✓	✓	Partially	Many
Disjoint Fields	×	×	✓	×	Partially	0
Field Malloc	×	×	✓	×	Partially	1
Sane Types	✓	×	×	×	Partially	1
No-captured Fields	✓	×	✓	×	Partially	1
Acyclic	×	×	✓	×	Partially	0

Table 3.2: Summary of analysis algorithms implemented in CAF.

Independently, one can classify analysis implementations as either *base* or *functor* implementations. Functor implementations (Array of Structures, Kill Flow, Callsite Depth-Combinator, Semi-Local Function, Unique Access Paths, Field Malloc, Sane Typing, Non-captured Fields) are designed to generate foreign premise queries, thus initiating collaboration with other implementations.

3.7 Formal Semantics

This section formalizes the semantics of the query language. This formalization builds upon a formalization of the small-step semantics of the compiler IR. At a high level, this formalization will,

- extend the small-step semantics into an *instrumentation semantics* which tracks a *loop context stack* and *def-use metadata* on each memory location;

- define notions of feasible paths and restricted feasible paths from the small-step relation; and,
- define the behavior of a sound analysis implementation in terms of the above.

The formalization starts with the Vellvm formalization of the semantics of the LLVM IR [95, 94]. Vellvm defines a small-step non-deterministic semantics called $\mathrm{LLVM_{ND}}$. $\mathrm{LLVM_{ND}}$ relates subsequent program states under a given program configuration:

$$\mathrm{mod}, g, \theta \vdash M, \overline{\Sigma} \longrightarrow_{\mathrm{ND}} M', \overline{\Sigma}'$$

The context mod, g, θ captures the static portion of an execution. The module mod lists the LLVM bitcode representation of the program, g assigns a distinct identifier value to each global variable, and θ assigns a distinct identifier value to each function. The program state $M, \overline{\Sigma}$ captures the dynamic portions of an execution. A memory state M represents a set of memory allocations (or, *blocks*), each implemented as a contiguous array of bytes. $\overline{\Sigma}$ represents a stack of invocation frames. Among other things, each frame includes a list $(c_0, \overline{c})$ representing the current instruction c_0 (i.e., program counter) and a continuation $\overline{c}$ (i.e., the remaining instruction in the block), and an assignment Δ of dynamic values to each of the frame's register temporaries (i.e., the register set).

The $\mathrm{LLVM_{ND}}$ semantics are *non-deterministic*; the step relation potentially relates a single machine state to multiple successor states. Vellvm introduces this non-determinism to deal with undefined program behaviors, such as LLVM's `undef` value. Otherwise, $\mathrm{LLVM_{ND}}$ semantics contain no big surprises. The model of memory is byte-oriented, and the semantics employ a flattening algorithm to marshal multi-byte values into and out of memory blocks. Memory cells retain their last-stored value. $\mathrm{LLVM_{ND}}$ represents memory addresses as a pair: a block identifier and an offset within that block.

3.7.1 The Instrumentation Semantics

This dissertation presents an *instrumentation semantics* as an extension of the $\mathrm{LLVM_{ND}}$ semantics. The instrumentation semantics builds upon Vellvm's formalization and additionally tracks loop context $\overline{li}$ and resource usage metadata $\overline{du}$ along program executions. We call it "instrumentation semantics" because this formalization codifies a loop-aware memory profiler very similar to LAMP [59].

The instrumentation semantics introduces a small-step relation of the form:

$$\mathrm{mod}, g, \theta \vdash M, \overline{\Sigma}, \overline{li}, \overline{du} \longrightarrow_{\mathrm{CAF}} M', \overline{\Sigma}', \overline{li}', \overline{du}'$$

Where $\overline{li}$ denotes a loop context stack and $\overline{du}$ denotes def-use metadata for each byte of allocated memory, both described below.

To track loop context and def-use metadata, the instrumentation semantics defines a non-deterministic step relation for memory operations (allocation, deallocation, load, store) and for control flow operations (conditional branch, unconditional branch, return). Other instructions do not affect the loop context or resource usage metadata; the rules corresponding to other instructions are lifted from the $\mathrm{LLVM_{ND}}$ semantics as follows:

$$\frac{\mathrm{mod}, g, \theta \vdash M, \overline{\Sigma} \longrightarrow_{\mathrm{ND}} M', \overline{\Sigma}'}{\mathrm{mod}, g, \theta \vdash M, \overline{\Sigma}, \overline{li}, \overline{du} \longrightarrow_{\mathrm{CAF}} M', \overline{\Sigma}', \overline{li}, \overline{du}}\mathrm{CAF\_LIFT\_DEFAULT}$$

Loop Context Stacks

The loop context stack captures an ordered sequence of loop contexts, representing the nested invocation of zero or more loops. Each loop context contains a loop (specified as a loop header basic block H) and an iteration number (specified as a monotonically increasing integer i):

$$\overline{li} ::= li, \overline{li} | []$$

$$li ::= (H, i)$$

A few functions manipulate these stacks. We use $\mathrm{length}(\overline{li})$ to denote the length of a context stack $\overline{li}$. Function $\mathrm{innermost}(H, \overline{li})$ finds the position of the innermost invocation of a loop H in the context stack, where position 0 is the bottom of the stack:

$$\begin{aligned}
\mathrm{innermost}(H, []) &= \bot \\
\mathrm{innermost}(H, ((H, i), \overline{li})) &= \mathrm{length}(\overline{li}) \\
\mathrm{innermost}(H, ((H', i), \overline{li})) &= \mathrm{innermost}(H, \overline{li}) \quad [H \neq H'].
\end{aligned}$$

Function $\mathrm{loopAt}(d, \overline{li})$ determines the loop active at position d in the loop stack $\overline{li}$:

$$\begin{aligned}
\mathrm{loopAt}(d, \overline{li}) &= \mathrm{loopAtRev}(\mathrm{length}(\overline{li}) - d - 1, \overline{li}) \\
\mathrm{loopAtRev}(d, []) &= \bot \\
\mathrm{loopAtRev}(0, ((H, i), \overline{li})) &= H \\
\mathrm{loopAtRev}(d, ((H, i), \overline{li})) &= \mathrm{loopAtRev}(d - 1, \overline{li}) \qquad [d > 0].
\end{aligned}$$

Function $\mathrm{iterationAt}(d, \overline{li})$ determines the iteration number at position d in $\overline{li}$:

$$\begin{aligned}
\mathrm{iterationAt}(d, \overline{li}) &= \mathrm{iterationAtRev}(\mathrm{length}(\overline{li}) - d - 1, \overline{li}) \\
\mathrm{iterationAtRev}(d, []) &= \bot \\
\mathrm{iterationAtRev}(0, ((H, i), \overline{li})) &= i \\
\mathrm{iterationAtRev}(d, ((H, i), \overline{li})) &= \mathrm{iterationAtRev}(d - 1, \overline{li}) \qquad [d > 0].
\end{aligned}$$

These loop context stacks disambiguate dynamic instances of static instructions. By combining a static instruction identifier with a loop context stack, we form a *contextualized operation*:

$$cop ::= c, \overline{li}$$

Memory def-use Metadata

A *resource* represents either the location of a single byte of memory or an externally visible resource (i.e., operating system side effect). Per Vellvm's construction, a memory location is a memory block identifier and an offset within that block:

$$rc ::= \text{blkid}, \text{offset} | \texttt{IO}$$

The instrumentation semantics will track the last definition of each resource, as well as a set of uses for each resource. A definition is simply a contextualized operation, indicating the last instruction to define that resource, or a bottom value indicating that the resource has not yet been defined:

$$def ::= cop | \bot$$

The set of uses is a set of contextualized operations which have used a resource:

$$\overline{use} ::= use, \overline{use} | []$$

$$use ::= cop$$

Finally, metadata represents a map from resources to definitions and use sets. The map is implemented as a list of associations $\overline{du}$,

$$\overline{du} ::= du, \overline{du} | []$$

$$du ::= (rc, def, \overline{use})$$

Semantics of Memory Operations

The instrumentation semantics updates the loop context stack and def-use metadata in response to memory allocation and deallocation, memory loads and stores, control flow, and procedure invocations. Memory operations affect the def-use metadata, but do not affect the loop context stack. Figure 3.9 defines the small-step semantics for `load`, `store`, `malloc`, and `free` operations.

A memory load instruction accepts a pointer operand val_p. Rule CAF_LD evaluates val_p with respect to the machine state to determine a set of possible dynamic values V_p. It chooses $v_p \in V_p$ non-deterministically. It updates metadata to record that the dynamic instance $(c_0, \overline{li})$ of the `load` instruction uses the resources located at v_p through $v_p +$ `sizeof`$(typ) - 1$.

A memory store instruction accepts two operands: a pointer val_p and a value to be stored val_s. Like load, it evaluates val_p and non-deterministically chooses a dynamic pointer v_p. It updates metadata to record that the dynamic instance $(c_0, \overline{li})$ of the `store` instruction defines the resources located at v_p through $v_p +$ `sizeof`$(typ) - 1$. It also clears the use-sets for those resources.

Allocation and deallocation are modeled as `store` instructions which overwrite the entire memory block. Figure 3.9 shows the semantics for `malloc` and `free`. Similar rules are necessary for `alloca` and the implicit deallocation of stack variables upon function return; those rules are not shown here.

Semantics of Control-Flow Operations

To maintain the loop context, the instrumentation semantics updates the loop context in response to control flow instructions: conditional branchs, unconditional branches, and return instructions. Figure 3.10 presents rules to maintain loop context upon reaching a branch instruction. Listed here are only those updated rules for a conditional branch that take its `true` successor. Other cases are very similar and are omitted.

$$\frac{\begin{array}{c}\mathrm{mod}, g, \theta \vdash M, \overline{\Sigma} \longrightarrow_{\mathrm{ND}} M, \overline{\Sigma}' \\ \overline{\Sigma} = (fid, l, (c_0, \overline{c}), tmn, \Delta, \alpha) \qquad c_0 = (id = \texttt{load}\, typ\, val_p\, align) \\ \mathbf{eval}_{\mathrm{ND}}(g, \Delta, val_p) = \lfloor V_p \rfloor \qquad v_p \in V_p \\ \overline{du}' = update\_ld(\overline{du}, v_p, \texttt{sizeof}(typ), (c_0, \overline{li}))\end{array}}{\mathrm{mod}, g, \theta \vdash M, \overline{\Sigma}, \overline{li}, \overline{du} \longrightarrow_{\mathrm{CAF}} M, \overline{\Sigma}', \overline{li}, \overline{du}'}\ \text{CAF\_LD}$$

$$\frac{\begin{array}{c}\mathrm{mod}, g, \theta \vdash M, \overline{\Sigma} \longrightarrow_{\mathrm{ND}} M', \overline{\Sigma}' \\ \overline{\Sigma} = (fid, l, (c_0, \overline{c}), tmn, \Delta, \alpha) \qquad c_0 = (\texttt{store}\, typ\, val_p\, val_s\, align) \\ \mathbf{eval}_{\mathrm{ND}}(g, \Delta, val_p) = \lfloor V_p \rfloor \qquad v_p \in V_p \\ \overline{du}' = update\_st(\overline{du}, v_p, \texttt{sizeof}(typ), (c_0, \overline{li}))\end{array}}{\mathrm{mod}, g, \theta \vdash M, \overline{\Sigma}, \overline{li}, \overline{du} \longrightarrow_{\mathrm{CAF}} M', \overline{\Sigma}', \overline{li}, \overline{du}'}\ \text{CAF\_ST}$$

$$\frac{\begin{array}{c}\mathrm{mod}, g, \theta \vdash M, \overline{\Sigma}_1 \longrightarrow_{\mathrm{ND}} M, \overline{\Sigma}_2 \\ \overline{\Sigma}_1 = ((fid, l_0, (c_0, \overline{c}), tmn, \Delta, \alpha), \overline{\Sigma}) \quad c_0 = (id = \texttt{malloc}\, typ\, val\, align) \\ \mathbf{eval}_{\mathrm{ND}}(g, \Delta, val) = \lfloor V_s \rfloor \qquad v_s \in V_s \\ \mathbf{eval}_{\mathrm{ND}}(g, \Delta, id) = \lfloor V_p \rfloor \qquad v_p \in V_p \\ \overline{du}' = update\_st(\overline{du}, v_p, v_s, (c_0, \overline{li}))\end{array}}{\mathrm{mod}, g, \theta \vdash M, \overline{\Sigma}_1, \overline{li}, \overline{du} \longrightarrow_{\mathrm{CAF}} M, \overline{\Sigma}_2, \overline{li}, \overline{du}'}\ \text{CAF\_MALLOC}$$

$$\frac{\begin{array}{c}\mathrm{mod}, g, \theta \vdash M, \overline{\Sigma}_1 \longrightarrow_{\mathrm{ND}} M, \overline{\Sigma}_2 \\ \overline{\Sigma}_1 = ((fid, l_0, (c_0, \overline{c}), tmn, \Delta, \alpha), \overline{\Sigma}) \qquad c_0 = (\texttt{free}\, typ\, val) \\ \mathbf{eval}_{\mathrm{ND}}(g, \Delta, val) = \lfloor V_p \rfloor \qquad v_p \in V_p \\ \overline{du}' = update\_st(\overline{du}, v_p, \texttt{sizeof}(typ), (c_0, \overline{li}))\end{array}}{\mathrm{mod}, g, \theta \vdash M, \overline{\Sigma}_1, \overline{li}, \overline{du} \longrightarrow_{\mathrm{CAF}} M, \overline{\Sigma}_2, \overline{li}, \overline{du}'}\ \text{CAF\_FREE}$$

Figure 3.9: Instrumentation semantics for memory operations.

$$\frac{\begin{array}{c}\mathrm{mod}, g, \theta \vdash M, \overline{\Sigma}_1 \longrightarrow_{\mathrm{ND}} M, \overline{\Sigma}_2 \\ \overline{\Sigma}_1 = ((fid, l_0, [], \texttt{br}\, v\, l_1\, l_2, \Delta, \alpha), \overline{\Sigma}) \quad \overline{\Sigma}_2 = ((fid, l_1, \overline{c}_1, tmn_1, \Delta', \alpha), \overline{\Sigma}) \\ \text{Edge } \langle l_0, l_1 \rangle \text{ enters loop H}\end{array}}{\mathrm{mod}, g, \theta \vdash M, \overline{\Sigma}_1, \overline{li}, \overline{du} \longrightarrow_{\mathrm{CAF}} M, \overline{\Sigma}_2, ((H, 0), \overline{li}), \overline{du}}\ \text{CAF\_BR\_ENTER}$$

$$\frac{\begin{array}{c}\mathrm{mod}, g, \theta \vdash M, \overline{\Sigma}_1 \longrightarrow_{\mathrm{ND}} M, \overline{\Sigma}_2 \\ \overline{\Sigma}_1 = ((fid, l_0, [], \texttt{br}\, v\, l_1\, l_2, \Delta, \alpha), \overline{\Sigma}) \quad \overline{\Sigma}_2 = ((fid, l_1, \overline{c}_1, tmn_1, \Delta', \alpha), \overline{\Sigma}) \\ \text{Edge } \langle l_0, l_1 \rangle \text{ is a backedge of loop H}\end{array}}{\mathrm{mod}, g, \theta \vdash M, \overline{\Sigma}_1, ((H, i), \overline{li}), \overline{du} \longrightarrow_{\mathrm{CAF}} M, \overline{\Sigma}_2, ((H, i+1), \overline{li}), \overline{du}}\ \text{CAF\_BR\_BACKEDGE}$$

$$\frac{\begin{array}{c}\mathrm{mod}, g, \theta \vdash M, \overline{\Sigma}_1 \longrightarrow_{\mathrm{ND}} M, \overline{\Sigma}_2 \\ \overline{\Sigma}_1 = ((fid, l_0, [], \texttt{br}\, v\, l_1\, l_2, \Delta, \alpha), \overline{\Sigma}) \quad \overline{\Sigma}_2 = ((fid, l_1, \overline{c}_1, tmn_1, \Delta', \alpha), \overline{\Sigma}) \\ \text{Edge } \langle l_0, l_1 \rangle \text{ exits loop H}\end{array}}{\mathrm{mod}, g, \theta \vdash M, \overline{\Sigma}_1, ((H, i), \overline{li}), \overline{du} \longrightarrow_{\mathrm{CAF}} M, \overline{\Sigma}_2, \overline{li}, \overline{du}}\ \text{CAF\_BR\_EXIT}$$

Figure 3.10: Instrumented semantics for the conditional branch $\texttt{br}\, v\, l_1\, l_2$ when $v = \texttt{true}$. Similar rules exist for the case when $v = \texttt{false}$, for unconditional branches, and for return instructions that exit one or more loops.

Suppose execution reaches a branch instruction $\texttt{br}\, v\, l_1\, l_2$ in basic block l_0, and suppose the branch transitions to successor block l_1. When the control flow edge $\langle l_0, l_1 \rangle$ enters a loop H, rule CAF_BR_ENTER pushes the new loop context $(H, 0)$ onto the loop context stack. When the $\langle l_0, l_1 \rangle$ traverses the backedge of loop H, rule CAF_BR_BACKEDGE increments the iteration number from (H, i) to $(H, i+1)$. When the $\langle l_0, l_1 \rangle$ exits a loop H, rule CAF_BR_EXIT pops the loop context from the stack.

3.7.2 Feasible Paths and Loop-Restrictions on Paths

The step relation $\longrightarrow_{\mathrm{CAF}}$ computes the effect of executing a single LLVM instruction. By extension, its transitive closure $\longrightarrow^*_{\mathrm{CAF}}$ relates machine states corresponding to endpoints of feasible paths. Figure 3.11 defines the $\longrightarrow^*_{\mathrm{CAF}}$ multistep relation as one or more single steps.

A loop H and an integral position d within the loop context stack specify a loop invo-

cation.[2] For any fixed loop invocation, we construct the $\longrightarrow_{\text{within}(H,d)}$ relation: a restriction of the $\longrightarrow_{\text{CAF}}$ relation that disallows any transition that would exit the invocation of loop H at height d in the loop context stack. Figure 3.12 provides a formal definition of this restriction. Specifically, $\longrightarrow_{\text{within}(H,d)}$ requires that the before and after states both feature a loop context that includes an invocation of loop H at position d.

The transitive closure $\longrightarrow^*_{\text{within}(H,d)}$ relates machines states corresponding to endpoints of those feasible paths which remain within an invocation of loop H at position d in the loop context stack. $\longrightarrow^*_{\text{within}(H,d)}$ is derived from $\longrightarrow_{\text{within}(H,d)}$ in a manner similar to how Figure 3.11 derives $\longrightarrow^*_{\text{CAF}}$ from $\longrightarrow_{\text{CAF}}$.

Next, we build some high-level multistep operators. For any fixed loop H and any two fixed instructions i_1, i_2, Figure 3.13 constructs the *connects-to-in* relation. Connects-to-in relates machine states found along feasible paths of execution which visit i_1 and then i_2 within an invocation d of loop H. Its definition is straightforward:

1. Execution starts from an initial state and reaches (via $\longrightarrow^*_{\text{CAF}}$) a dynamic instance of instruction i_1;

2. after executing i_1 (via $\longrightarrow_{\text{within}(H,d)}$), the machine is in state $S_{after1} = (M_{after1}, \overline{\Sigma}_{after1}, \overline{li}_{after1}, \overline{du}_{after1})$;

3. execution continues (via $\longrightarrow^*_{\text{within}(H,d)}$) to a dynamic instance of instruction i_2; and,

4. after executing i_2 (via $\longrightarrow_{\text{within}(H,d)}$, the machine is in state $S_{after2} = (M_{after2}, \overline{\Sigma}_{after2}, \overline{li}_{after2}, \overline{du}_{after2})$.

Figure 3.14 constructs $\longrightarrow_{\text{same}(H,i_1,i_2)}$ and $\longrightarrow_{\text{cross}(H,i_1,i_2)}$. Both of these operators relate machine states S_{after1} and S_{after2} which occur on certain paths of execution. The difference is whether the path from i_1 to i_2 stays within the same iteration of H (as in $\longrightarrow_{\text{same}(H,i_1,i_2)}$) or crosses the loop backedge of H (as in $\longrightarrow_{\text{cross}(H,i_1,i_2)}$).

[2]Because of recursive procedures, one loop H may experience several simultaneous invocations. This semantics employs the position d to distinguish these cases.

$$\frac{\mathrm{mod}, g, \theta \vdash M_1, \overline{\Sigma}_1, \overline{li}_1, \overline{du}_1 \longrightarrow_{\mathrm{CAF}} M_2, \overline{\Sigma}_2, \overline{li}_2, \overline{du}_2}{\mathrm{mod}, g, \theta \vdash M_1, \overline{\Sigma}_1, \overline{li}_1, \overline{du}_1 \longrightarrow^*_{\mathrm{CAF}} M_2, \overline{\Sigma}_2, \overline{li}_2, \overline{du}_2}\text{CAF\_TC1}$$

$$\frac{\begin{array}{l}\mathrm{mod}, g, \theta \vdash M_1, \overline{\Sigma}_1, \overline{li}_1, \overline{du}_1 \longrightarrow^*_{\mathrm{CAF}} M_2, \overline{\Sigma}_2, \overline{li}_2, \overline{du}_2 \\ \mathrm{mod}, g, \theta \vdash M_2, \overline{\Sigma}_2, \overline{li}_2, \overline{du}_2 \longrightarrow_{\mathrm{CAF}} M_3, \overline{\Sigma}_3, \overline{li}_3, \overline{du}_3\end{array}}{\mathrm{mod}, g, \theta \vdash M_1, \overline{\Sigma}_1, \overline{li}_1, \overline{du}_1 \longrightarrow^*_{\mathrm{CAF}} M_3, \overline{\Sigma}_3, \overline{li}_3, \overline{du}_3}\text{CAF\_TC2}$$

Figure 3.11: The multistep relation $\longrightarrow^*_{\mathrm{CAF}}$ denotes one or more execution steps and is defined as one or more single steps. It relates machine states corresponding to the endpoints of feasible paths.

$$\frac{\begin{array}{c}\mathrm{mod}, g, \theta \vdash M_1, \overline{\Sigma}_1, \overline{li}_1, \overline{du}_1 \longrightarrow_{\mathrm{CAF}} M_2, \overline{\Sigma}_2, \overline{li}_2, \overline{du}_2 \\ \mathrm{loopAt}(d, \overline{li_1}) = H \qquad\qquad \mathrm{loopAt}(d, \overline{li_2}) = H\end{array}}{\mathrm{mod}, g, \theta \vdash M_1, \overline{\Sigma}_1, \overline{li}_1, \overline{du}_1 \longrightarrow_{\mathrm{within}(H,d)} M_2, \overline{\Sigma}_2, \overline{li}_2, \overline{du}_2}\text{CAF\_WITHIN}$$

Figure 3.12: The Path-restricted small-step operator $\longrightarrow_{\mathrm{within}(H,d)}$ relates machine states such that execution remains within the same invocation of loop H at depth d.

3.7.3 The `modref_ii`$(i_1, \texttt{Same}, i_2, H)$ Query

Figure 3.15 defines the semantics of the `modref_ii`$(i_1, \texttt{Same}, i_2, H)$ query.

All rules follow a common form. The first hypothesis selects intra-iteration paths of execution. Subsequent hypotheses handle specific cases. The first two rules test whether instruction i_1 defines a resource that instruction i_2 will read (CAF_MRS_MOD1) or write (CAF_MRS_MOD2). The second two rules test whether instruction i_1 uses a resource that instruction i_2 will read (CAF_MRS_REF1) or write (CAF_MRS_REF2).

3.7.4 The `modref_ii`$(i_1, \texttt{Before}, i_2, H)$ Query

Figure 3.16 defines the semantics of the `modref_ii`$(i_1, \texttt{Before}, i_2, H)$ query. The structure of these definitions corresponds to those for `modref_ii`$(i_1, \texttt{Same}, i_2, H)$, substituting $\longrightarrow_{\mathrm{cross}(H,i_1,i_2)}$ for $\longrightarrow_{\mathrm{same}(H,i_1,i_2)}$.

The first two rules test whether instruction i_1 defines a resource that instruction i_2 will read (CAF_MRB_MOD1) or write (CAF_MRB_MOD2). The second two rules test

$$\frac{\begin{array}{c}\text{mod}, g, \theta \vdash S_0 \longrightarrow^*_{\text{CAF}} S_1 \\ \text{mod}, g, \theta \vdash S_1 \longrightarrow_{\text{within}(H,d)} S_{after1} \\ \text{mod}, g, \theta \vdash S_{after1} \longrightarrow^*_{\text{within}(H,d)} S_2 \\ \text{mod}, g, \theta \vdash S_2 \longrightarrow_{\text{within}(H,d)} S_{after2} \\ S_1 = M_1, \overline{\Sigma}_1, \overline{li}_1, \overline{du}_1 \\ \overline{\Sigma}_1 = ((fid_1, l_1, (i_1, \overline{c}_1), tmn_1, \Delta_1, \alpha_1), \overline{\Sigma}'_1) \\ S_2 = M_2, \overline{\Sigma}_2, \overline{li}_2, \overline{du}_2 \\ \overline{\Sigma}_2 = ((fid_2, l_2, (i_2, \overline{c}_2), tmn_2, \Delta_2, \alpha_2), \overline{\Sigma}'_2)\end{array}}{\text{mod}, g, \theta \vdash (S_1, S_{after1}, S_2, S_{after2}) \text{ connect } i_1 \text{ to } i_2 \text{ in } H, d}\text{CAF\_CONNECT}$$

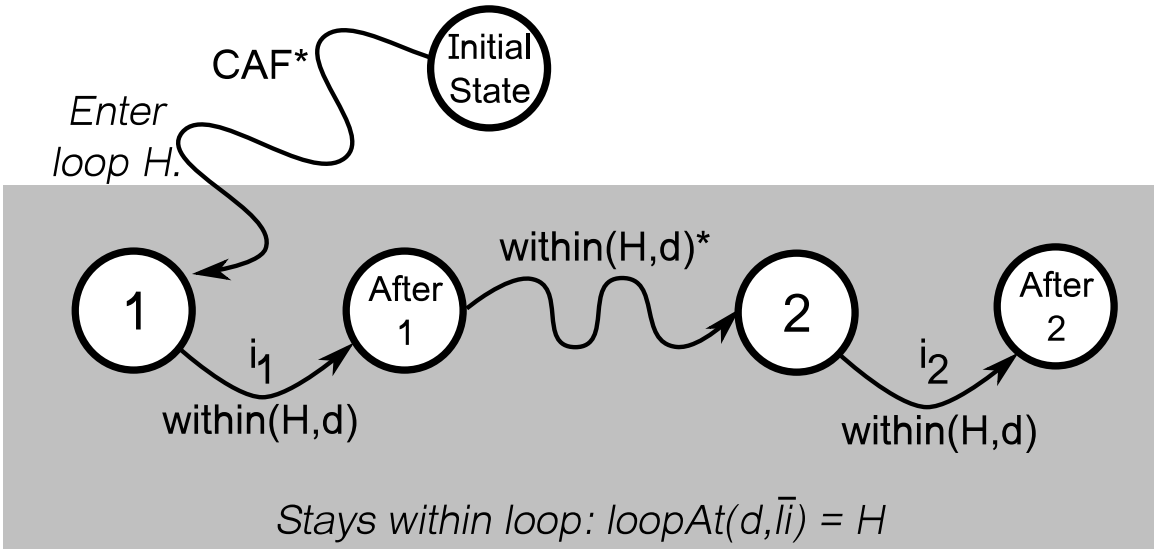


Figure 3.13: *(above)* Rule CAF_CONNECT defines the connects-to-in relation. For any fixed instructions i_1, i_2 and any fixed loop invocation H, d, this relates relates four machine states along feasible paths that include i_1 and i_2. State 1 occurs immediately before instruction i_1, and state After1 immediately follows i_1. State 2 occurs immediately before instruction i_2 and state After2 immediately follows i_2. All four states occur within loop invocation (H, d). *(below)* An illustration of the rule CAF_CONNECT, showing machine states as circles and arrows representing multistep ($\longrightarrow^*_{\text{CAF}}$, $\longrightarrow^*_{\text{within}(H,d)}$) and single step ($\longrightarrow_{\text{within}(H,d)}$) relations.

$$\frac{\begin{array}{c}\text{mod}, g, \theta \vdash (S_1, S_{after1}, S_2, S_{after2}) \text{ connect } i_1 \text{ to } i_2 \text{ in } H, d \\ S_1 = M_1, \overline{\Sigma}_1, \overline{li}_1, \overline{du}_1; \quad S_{after1} = M_{after1}, \overline{\Sigma}_{after1}, \overline{li}_{after1}, \overline{du}_{after1} \\ S_2 = M_2, \overline{\Sigma}_2, \overline{li}_2, \overline{du}_2; \quad S_{after2} = M_{after2}, \overline{\Sigma}_{after2}, \overline{li}_{after2}, \overline{du}_{after2} \\ \text{iterationAt}(d, \overline{li}_1) = \text{iterationAt}(d, \overline{li}_2) \\ d = \min_{\overline{li} \in \{\overline{li}_1, \overline{li}_{after1}, \overline{li}_2, \overline{li}_{after2}\}} \text{innermost}(H, \overline{li})\end{array}}{\text{mod}, g, \theta \vdash S_{after1} \longrightarrow_{\text{same}(H, i_1, i_2)} S_{after2}} \text{CAF\_SAME}$$

$$\frac{\begin{array}{c}\text{mod}, g, \theta \vdash (S_1, S_{after1}, S_2, S_{after2}) \text{ connect } i_1 \text{ to } i_2 \text{ in } H, d \\ S_1 = M_1, \overline{\Sigma}_1, \overline{li}_1, \overline{du}_1; \quad S_{after1} = M_{after1}, \overline{\Sigma}_{after1}, \overline{li}_{after1}, \overline{du}_{after1} \\ S_2 = M_2, \overline{\Sigma}_2, \overline{li}_2, \overline{du}_2; \quad S_{after2} = M_{after2}, \overline{\Sigma}_{after2}, \overline{li}_{after2}, \overline{du}_{after2} \\ \text{iterationAt}(d, \overline{li}_1) < \text{iterationAt}(d, \overline{li}_2) \\ d = \min_{\overline{li} \in \{\overline{li}_1, \overline{li}_{after1}, \overline{li}_2, \overline{li}_{after2}\}} \text{innermost}(H, \overline{li})\end{array}}{\text{mod}, g, \theta \vdash S_{after1} \longrightarrow_{\text{cross}(H, i_1, i_2)} S_{after2}} \text{CAF\_CROSS}$$

Figure 3.14: Two restricted multistep relations. $\longrightarrow_{\text{same}(H,i_1,i_2)}$ relates machine states which along feasible paths that visit instruction i_1 and i_2 within a common iteration of the innermost invocation of loop H. $\longrightarrow_{\text{cross}(H,i_1,i_2)}$ relates machine states along feasible paths that visit instruction i_1 and i_2 within different iterations of the innermost invocation of loop H.

whether instruction i_1 uses a resource that instruction i_2 will read (CAF_MRB_REF1) or write (CAF_MRB_REF2).

3.8 Discussion

This chapter introduced collaboration, factored development, details of several implementations, and a formalization of the query language.

3.8.1 Development of Factored Analysis Algorithms

Because ensembles support collaboration, developers may modularize the development of analysis algorithms through *factorization*. Instead of increasingly complicated algorithms which incorporate additional types of reasoning, factorization achieves precision through many simple algorithms. Each algorithm disproves queries within its core competence

$$\frac{\begin{array}{c}\mathrm{mod}, g, \theta \vdash S_{after1} \longrightarrow_{\mathrm{same}(H,i_1,i_2)} S_{after2} \\ S_{after2} = M_{after2}, \overline{\Sigma}_{after2}, \overline{li}_{after2}, \overline{du}_{after2} \\ (rc_2, def_2, \overline{use}_2) \in \overline{du}_{after2} \\ def_2 = (i_1, \overline{li}_1) \\ (i_2, \overline{li}_2) \in \overline{use}_2\end{array}}{\texttt{modref\_ii}(i_1, \texttt{Same}, i_2, H) \text{ responds } \texttt{Mod} \text{ or } \texttt{ModRef}.} \text{CAF\_MRS\_MOD1}$$

$$\frac{\begin{array}{c}\mathrm{mod}, g, \theta \vdash S_{after1} \longrightarrow_{\mathrm{same}(H,i_1,i_2)} S_{after2} \\ S_{after1} = M_{after1}, \overline{\Sigma}_{after1}, \overline{li}_{after1}, \overline{du}_{after1} \\ (rc_{1-2}, def_1, \overline{use}_1) \in \overline{du}_{after1} \\ def_1 = (i_1, \overline{li}_1) \\ S_{after2} = M_{after2}, \overline{\Sigma}_{after2}, \overline{li}_{after2}, \overline{du}_{after2} \\ (rc_{1-2}, def_2, \overline{use}_2) \in \overline{du}_{after2} \\ def_2 = (i_2, \overline{li}_2)\end{array}}{\texttt{modref\_ii}(i_1, \texttt{Same}, i_2, H) \text{ responds } \texttt{Mod} \text{ or } \texttt{ModRef}.} \text{CAF\_MRS\_MOD2}$$

$$\frac{\begin{array}{c}\mathrm{mod}, g, \theta \vdash S_{after1} \longrightarrow_{\mathrm{same}(H,i_1,i_2)} S_{after2} \\ S_{after1} = M_{after1}, \overline{\Sigma}_{after1}, \overline{li}_{after1}, \overline{du}_{after1} \\ (rc_{1-2}, def_1, \overline{use}_1) \in \overline{du}_{after1} \\ (i_1, \overline{li}_1) \in \overline{use}_1 \\ S_{after2} = M_{after2}, \overline{\Sigma}_{after2}, \overline{li}_{after2}, \overline{du}_{after2} \\ (rc_{1-2}, def_2, \overline{use}_2) \in \overline{du}_{after2} \\ (i_2, \overline{li}_2) \in \overline{use}_2\end{array}}{\texttt{modref\_ii}(i_1, \texttt{Same}, i_2, H) \text{ responds } \texttt{Ref} \text{ or } \texttt{ModRef}.} \text{CAF\_MRS\_REF1}$$

$$\frac{\begin{array}{c}\mathrm{mod}, g, \theta \vdash S_{after1} \longrightarrow_{\mathrm{same}(H,i_1,i_2)} S_{after2} \\ S_{after1} = M_{after1}, \overline{\Sigma}_{after1}, \overline{li}_{after1}, \overline{du}_{after1} \\ (rc_{1-2}, def_1, \overline{use}_1) \in \overline{du}_{after1} \\ (i_1, \overline{li}_1) \in \overline{use}_1 \\ S_{after2} = M_{after2}, \overline{\Sigma}_{after2}, \overline{li}_{after2}, \overline{du}_{after2} \\ (rc_{1-2}, def_2, \overline{use}_2) \in \overline{du}_{after2} \\ def_2 = (i_2, \overline{li}_2)\end{array}}{\texttt{modref\_ii}(i_1, \texttt{Same}, i_2, H) \text{ responds } \texttt{Ref} \text{ or } \texttt{ModRef}.} \text{CAF\_MRS\_REF2}$$

Figure 3.15: The semantics of $\texttt{modref\_ii}(i_1, \texttt{Same}, i_2, H)$.

$$\frac{\begin{array}{c}\mathrm{mod}, g, \theta \vdash S_{after1} \longrightarrow_{\mathrm{cross}(H,i_1,i_2)} S_{after2} \\ S_{after2} = M_{after2}, \overline{\Sigma}_{after2}, \overline{li}_{after2}, \overline{du}_{after2} \\ (rc_2, def_2, \overline{use}_2) \in \overline{du}_{after2} \\ def_2 = (i_1, \overline{li}_1) \\ (i_2, \overline{li}_2) \in \overline{use}_2\end{array}}{\texttt{modref\_ii}(i_1, \texttt{Before}, i_2, H) \text{ responds } \texttt{Mod} \text{ or } \texttt{ModRef}.} \text{CAF\_MRB\_MOD1}$$

$$\frac{\begin{array}{c}\mathrm{mod}, g, \theta \vdash S_{after1} \longrightarrow_{\mathrm{cross}(H,i_1,i_2)} S_{after2} \\ S_{after1} = M_{after1}, \overline{\Sigma}_{after1}, \overline{li}_{after1}, \overline{du}_{after1} \\ (rc_{1-2}, def_1, \overline{use}_1) \in \overline{du}_{after1} \\ def_1 = (i_1, \overline{li}_1) \\ S_{after2} = M_{after2}, \overline{\Sigma}_{after2}, \overline{li}_{after2}, \overline{du}_{after2} \\ (rc_{1-2}, def_2, \overline{use}_2) \in \overline{du}_{after2} \\ def_2 = (i_2, \overline{li}_2)\end{array}}{\texttt{modref\_ii}(i_1, \texttt{Before}, i_2, H) \text{ responds } \texttt{Mod} \text{ or } \texttt{ModRef}.} \text{CAF\_MRB\_MOD2}$$

$$\frac{\begin{array}{c}\mathrm{mod}, g, \theta \vdash S_{after1} \longrightarrow_{\mathrm{cross}(H,i_1,i_2)} S_{after2} \\ S_{after1} = M_{after1}, \overline{\Sigma}_{after1}, \overline{li}_{after1}, \overline{du}_{after1} \\ (rc_{1-2}, def_1, \overline{use}_1) \in \overline{du}_{after1} \\ (i_1, \overline{li}_1) \in \overline{use}_1 \\ S_{after2} = M_{after2}, \overline{\Sigma}_{after2}, \overline{li}_{after2}, \overline{du}_{after2} \\ (rc_{1-2}, def_2, \overline{use}_2) \in \overline{du}_{after2} \\ (i_2, \overline{li}_2) \in \overline{use}_2\end{array}}{\texttt{modref\_ii}(i_1, \texttt{Before}, i_2, H) \text{ responds } \texttt{Ref} \text{ or } \texttt{ModRef}.} \text{CAF\_MRB\_REF1}$$

$$\frac{\begin{array}{c}\mathrm{mod}, g, \theta \vdash S_{after1} \longrightarrow_{\mathrm{cross}(H,i_1,i_2)} S_{after2} \\ S_{after1} = M_{after1}, \overline{\Sigma}_{after1}, \overline{li}_{after1}, \overline{du}_{after1} \\ (rc_{1-2}, def_1, \overline{use}_1) \in \overline{du}_{after1} \\ (i_1, \overline{li}_1) \in \overline{use}_1 \\ S_{after2} = M_{after2}, \overline{\Sigma}_{after2}, \overline{li}_{after2}, \overline{du}_{after2} \\ (rc_{1-2}, def_2, \overline{use}_2) \in \overline{du}_{after2} \\ def_2 = (i_2, \overline{li}_2)\end{array}}{\texttt{modref\_ii}(i_1, \texttt{Before}, i_2, H) \text{ responds } \texttt{Ref} \text{ or } \texttt{ModRef}.} \text{CAF\_MRB\_REF2}$$

Figure 3.16: The semantics of $\texttt{modref\_ii}(i_1, \texttt{Before}, i_2, H)$.

and assumes other algorithms provide the necessary diversity of logic to solve its foreign premises. Factored algorithms are developed independently without requiring knowledge of others. Factorization enables developers to easily extend algorithm precision according to the needs of a client.

While developing compiler transformations, one often observes the compiler acting conservatively due to analysis imprecision. Such cases indicate a deficiency of the ensemble, and represent an untapped niche in the design space for a new algorithm. This section describes a process to develop new algorithms which are precise in these cases without requiring developer knowledge of the rest of the ensemble.

Developing algorithms in the proposed framework is no more complicated than developing a monolithic algorithm because composability allows the developer to remain largely ignorant of other algorithms in the ensemble. This proposal frequently *simplifies* the process by addressing smaller algorithms in isolation. This proposal does not simplify termination arguments, though in practice they are simple.

The developer first enumerates dependence queries with imprecise results, either manually or by using an automated tool to compare static analysis to profiling. The developer confirms that each query is imprecise by arguing *why* its corresponding dependence cannot manifest. The developer generalizes this argument into an algorithm to recognize such queries and report independence. Algorithms discovered through this process target queries which affect the client, focus on common programming idioms instead of the general case, and are largely *orthogonal* (see Section 6.1.3) to the ensemble.

3.8.2 Generalization to other Analysis Problems

The problem of dependence analysis primarily motivates the design of the CAF. However, the CAF is general enough to accommodate different undecidable problems by employing alternative query languages. Other details, such assembly of analysis implementations into an ensemble, topping and chaining, and termination guarantees extend to new problem

domains without change.

3.8.3 Marrying Dependence Analysis with Speculation

This dissertation believes that speculation and dependence analysis are so intertwined that they should be considered together. Chapter 5 extends CAF to support collaboration with *speculation modules* to achieve *speculative dependence identification*.

Chapter 4

The *Fast* $\mathrm{DAG}_{\mathrm{SCC}}$ Algorithm

"I don't care how much power, brilliance or energy you have,
if you don't harness it and focus it on a specific target, and hold it there
you're never going to accomplish as much as your ability warrants."
–Zig Ziglar

Users expect compilers to be fast [83] and to generate efficient machine code. However, aggressive compiler optimizations are sensitive to the quality of dependence identification [15, 27, 71, 74, 88], and precise dependence identification tends to be expensive [22, 32] and scale poorly [33, 60]. These constraints are mutually detrimental.

This chapter presents the Fast $\mathrm{DAG}_{\mathrm{SCC}}$ Algorithm which reduces analysis time while maintaining the same analysis precision. Specifically, the Fast $\mathrm{DAG}_{\mathrm{SCC}}$ Algorithm allows the compiler to compute the $\mathrm{DAG}_{\mathrm{SCC}}$ from a demand-driven dependence identification framework, such as the CAF presented in Chapter 3. By excluding certain dependence queries which cannot affect the $\mathrm{DAG}_{\mathrm{SCC}}$ structure, and which consequently have no effect on optimizing transformations, the Fast $\mathrm{DAG}_{\mathrm{SCC}}$ Algorithm halves analysis burden and achieves a comparable reduction in compilation time.

To be clear, the Fast $\mathrm{DAG}_{\mathrm{SCC}}$ algorithm is not an improvement over Tarjan's Algorithm. In fact, Tarjan's algorithm is optimal when a graph is known a priori. In a compiler, the

graph is not known until dependence analysis has computed the graph. Fast $\mathrm{DAG_{SCC}}$ is actually an improved algorithm for graph discovery while of computing the $\mathrm{DAG_{SCC}}$.

4.1 Background

In a Program Dependence Graph (PDG), each instruction in the scope is represented as a vertex. Edges represent control and data dependences. Figure 4.3(a) shows an example of a PDG.

The Directed Acyclic Graph of the Strongly Connected Components ($\mathrm{DAG_{SCC}}$) is simplification the PDG which represents dependence cycles explicitly. The $\mathrm{DAG_{SCC}}$ partitions the scope's instructions into strongly connected components: sets of instructions which are bi-connected by a dependence cycle, or singleton sets of instructions which do not partake in a dependence cycle. The $\mathrm{DAG_{SCC}}$ represents each block of that partition as a vertex and $\mathrm{DAG_{SCC}}$ edges relate components c_1, c_2 iff they contain instructions $i_1 \in c_1, i_2 \in c_2$ such that (i_1, i_2) is an edge in the PDG. The $\mathrm{DAG_{SCC}}$ contains less information than the PDG since it does not represent the many PDG edges among instructions assigned to a common component. The $\mathrm{DAG_{SCC}}$ is an ideal representation for certain scheduling algorithms including the DSWP-family of thread extraction techniques [71, 74, 87].

To construct the $\mathrm{DAG_{SCC}}$ of a loop, a naïve compiler considers the presence or absence of a dependence edge between every pair of vertices (operations) to compute a PDG, then identifies cycles in the PDG and condenses them into a $\mathrm{DAG_{SCC}}$. However, not all dependences in the PDG contribute equally to the structure of the $\mathrm{DAG_{SCC}}$. Once a PDG is partially computed, some edges have no marginal value since they do not affect the structure of the $\mathrm{DAG_{SCC}}$ and thus cannot affect the answer to optimization questions. By eliminating these redundant dependence edges, a compiler computes the $\mathrm{DAG_{SCC}}$ with fewer dependence queries in less time. Compiler authors may spend these savings on costlier analyses in pursuit of aggressive optimization.

An ideal algorithm would perform queries only for those edges found in a transitive reduction of the PDG (to *join* components), as well as queries to ensure the absence of back edges (to *separate* components). This, however, leads to a problem: the compiler does not know the PDG *a priori*, and so it cannot distinguish redundant edges from constructive ones. Instead, this dissertation proposes an approximation of that ideal.

The top half of Figure 4.1 illustrates one class of redundant dependences: edges that order two vertices whose components are already ordered. This is a large class of dependences, which grows quadratically in the number of components and quadratically in component size. Across SPEC CPU2006, empirical study indicates that two-thirds of all loops have 5–968 SCCs and two-thirds of all components have 8.4–1118.0 vertices.

Another class of redundant dependences are illustrated in the bottom half Figure 4.1: edges within a component other than a minimum cycle that spans the component. This class grows quadratically in component size and linearly in the number of components.

The only dependences which contribute to finding the condensation graph are the class which join separate components, demonstrated in Figure 4.2. These grow quadratically in the size of components and quadratically in the number of components. Although this is a large class, any one dependence between a pair of components will constrain the entire component. Conversely, the absence of these dependences also has value, since only after analysis returns *negative* results can the algorithm confidently report that the separate components are separate.

By periodically interrupting PDG construction to recompute strongly connected components, the proposed algorithm identifies dependence edges and eliminates dependence queries which are definitely in classes (d) and (e) while focusing on those dependence queries which seem to be within class (f). This approach is informed by the following heuristic: if the compiler can build large components quickly, it can safely exclude more edges. Further, this technique performs more computation to actively search for opportunities to elide queries. This strategy will not be faster in the worst case since the overhead

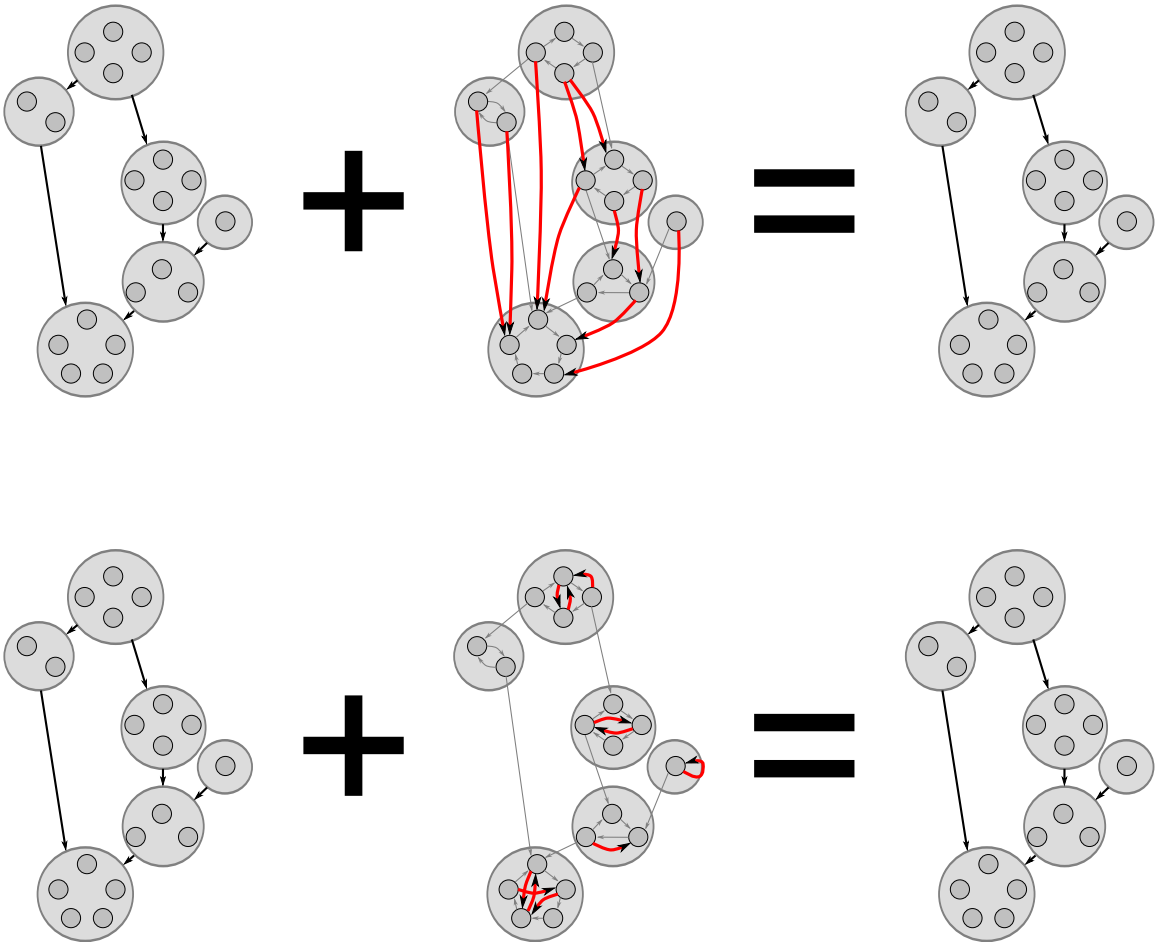


Figure 4.1: (*above*) The new red edges redundantly order components. They do not change the condensation, thus have no marginal value; (*below*) The new red edges remain within a component. They do not change the condensation, thus have no marginal value

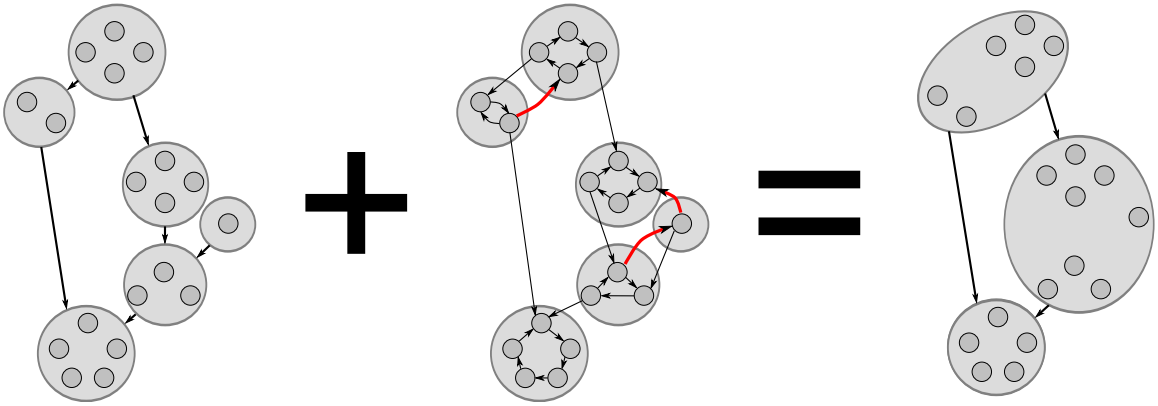


Figure 4.2: The new red edges violate the previous topological ordering. They are valuable since they invalidate the topological ordering and may change the condensation.

of recomputing components may overwhelm the benefits for loops with a very low average component size. However, the common case is more amenable to this strategy; experiments show that the proposed method is faster for all but 14 of 366 loops.

Control dependences represent cases where one instruction may prevent another instruction from executing; for instance, an `if`-statement controls its `then`- and `else`-clauses. Data dependences represent the flow of data between instructions. We distinguish *register* data dependences from *memory* data dependences. Register and control dependences are computed quickly in practice.

`Query`$(v_1.\text{inst}, v_2.\text{inst}, \text{type})$ denote a demand-driven dependence analysis query into the collaborative analysis framework. It determines whether there is a memory dependence from the instruction associated with vertex v_1 to the instruction associated with vertex v_2; type is either *Loop-Carried* or *Intra-Iteration*.

In the algorithms below, `TarjanSCC` refers to Tarjan's Algorithm for Strongly Connected Components [82]. Tarjan's algorithm reports SCC structure as well as a topological sort of those components and runs in time linear in the number of vertices and edges.

4.2 Baseline Algorithm

The baseline algorithm (Algorithm 1) builds a full PDG, including all register, control and memory dependences. To find memory dependences, it queries every pair of vertices (corresponding to instructions in the IR) which access memory to determine if there is a loop-carried or intra-iteration memory dependence. It then invokes Tarjan's algorithm to find the strongly connected components of that PDG, and condenses those components into a DAG_{SCC}.

Algorithm 1: Baseline computeDagScc(V)

```
let E := computeRegisterDeps(V) ∪ computeControlDeps(V);
foreach vertex v_src ∈ V which accesses memory do
    foreach vertex v_dst ∈ V which accesses memory do
        if Query(v_src.inst, v_dst.inst, Loop-Carried) then
            let E := E ∪ {⟨v_src, v_dst, Loop-Carried⟩};
        end
        if Query(v_src.inst, v_dst.inst, Intra-Iteration) then
            let E := E ∪ {⟨v_src, v_dst, Intra-Iteration⟩};
        end
    end
end
return TarjanSCC(V, E);
```

4.3 Client-Agnostic Algorithm

Algorithm 2 lists the client-agnostic version of the Fast $\mathrm{DAG_{SCC}}$ algorithm. The client-agnostic method starts by computing register and control dependences. This yields a PDG which is only partially computed since it lacks memory dependences. Next, it performs queries only between the vertices of select components in `withTheGrain` and `againstTheGrain`. These queries correspond to those dependence edges which most quickly merge components into larger components. This leads to a savings in the number of memory dependence queries since dependences between vertices in a common component cannot further constrain the $\mathrm{DAG_{SCC}}$.

The routine `withTheGrain` (Algorithm 3) considers pairs of components c_{early} and c_{late} where c_{early} appears *before* c_{late} in the topological sorting of components. `withTheGrain` exploits the feature that Tarjan's algorithm provides a topological sorting of the components with no additional computation. Figure 4.3(a) shows a topological sort. `withTheGrain` only performs queries that flow along topological order (i.e. from c_{early} to c_{late}), and only between components that are not already immediately ordered. Such queries neither cause separate components to merge, nor invalidate the topological sorting of components, as illustrated in Figure 4.3(b).

Algorithm 2: Client-Agnostic computeDagScc(V)

let E $\coloneqq$ computeRegisterDeps(V) $\cup$ computeControlDeps(V) ;
let TopSort$_0$ $\coloneqq$ TarjanSCC(V, E) ;
• *// (Point X)*
let E_0 $\coloneqq$ E $\cup$ withTheGrain(E, TopSort$_0$) ;
for *i = 1 to* ∞ **do**
  • *// (Point Y)*
  let E$'$ $\coloneqq$ againstTheGrain(TopSort$_{i-1}$) ;
  if $E' = \emptyset$ **then**
    return TopSort$_{i-1}$;
  end
  let E$'_i$ $\coloneqq$ E$_{i-1}$ $\cup$ E$'$;
  let TopSort$_i$ $\coloneqq$ TarjanSCC$(V,$ E$'_i)$;
  let E$_i$ $\coloneqq$ E$'_i$ $\cup$ withTheGrain(E$'_i$, TopSort$_i$) ;
end

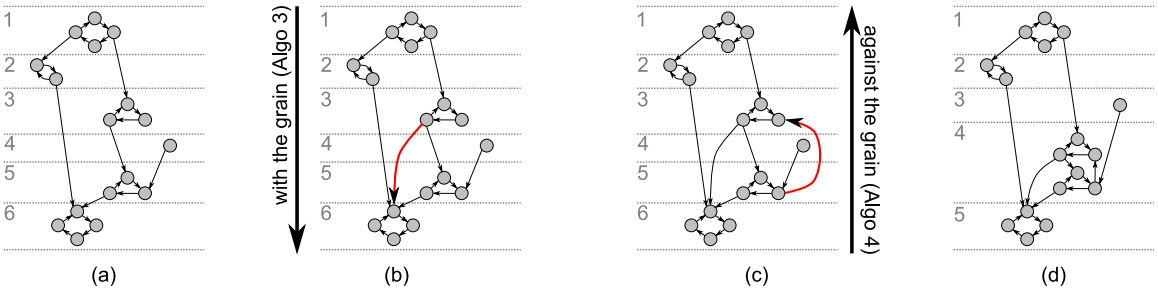


Figure 4.3: A partially computed PDG. (a) Topological sort (grey lines) imposes a total order on the partially ordered components. (b) `withTheGrain` (Algorithm 3) performs queries to discover edges between components with increasing position in the topological sort. Such edges neither cause SCCs to merge nor invalidate the topological sort. Here, a new edge is discovered from component three to six. (c) `againstTheGrain` (Algorithm 4) performs queries to discover edges between components with decreasing position. Here, a new edge is discovered from component five to three. (d) When `againstTheGrain` discovers new edges the topological sort is invalidated and components may merge.

Algorithm 3: withTheGrain(E_0, TopSort)

```
let E′ := ∅;
let N := size(TopSort);
for i = N-1 down to 0 do
    let c_late := TopSort(i);
    for j = i-1 down to 0 do
        let c_early := TopSort(j);
        if ¬hasEdge(c_early, c_late, E_0) then
            let E′ := E′ ∪ findOneEdge(c_early, c_late);
        end
    end
end
return E' ;
```

The routine `againstTheGrain` (Algorithm 4) searches for dependences between pairs of components. Unlike `withTheGrain`, `againstTheGrain` only performs queries which may add edges that violate topological sort order, i.e. those from a vertex in a component c_{late} to a vertex in a topologically-earlier c_{early}. The rationale is that such queries quickly form larger components (Figure 4.3(c)). Large components have a compounding effect, further reducing the number of queries performed later. This routine performs enough queries to test every absence of an edge if none exists, allowing the algorithm to report that two components are separate.

Algorithm 4: againstTheGrain(TopSort)

```
let E := ∅;
let N := size(TopSort);
for i = N-1 down to 0 do
    let c_late := TopSort(i);
    for j = i-1 down to 0 do
        let c_early := TopSort(j);
        let E′ := findOneEdge(c_late, c_early);
        let E := E ∪ E′;
        if E′ ≠ ∅ then
            break;
        end
    end
end
return E;
```

The routine `findOneEdge` (Algorithm 5) performs queries from a source component to destination component. It stops after it finds the first edge between them since additional edges would order those two components redundantly.

Algorithm 5: findOneEdge(c_{src}, c_{dst})

```
foreach vertex v_src ∈ c_src which accesses memory do
    foreach vertex v_dst ∈ c_dst which accesses memory do
        if Query(v_src.inst, v_dst.inst, Loop-Carried) then
            return {⟨v_src, v_dst, Loop-Carried⟩};
        end
    end
end
foreach vertex v_src ∈ c_src which accesses memory do
    foreach vertex v_dst ∈ c_dst which accesses memory do
        if Query(v_src.inst, v_dst.inst, Intra-Iteration) then
            return {⟨v_src, v_dst, Intra-Iteration⟩};
        end
    end
end
return ∅;
```

4.4 Extensions for PS-DSWP

The $\mathrm{DAG}_{\mathrm{SCC}}$ guides clients such as DSWP [74] or loop fission [6, 43]. Some clients want more information than the $\mathrm{DAG}_{\mathrm{SCC}}$ offers. The proposed algorithm may be extended to needs of particular clients. Despite these additional requirements, one can implement these extensions while achieving comparable performance improvements over the baseline. Two dimensions characterize client-specific extensions of the algorithm: additional requirements of dependence information and opportunities to abort early.

Parallel Stage Decoupled Software Pipelining (PS-DSWP) is an illustrative example of such a client. PS-DSWP is an automatic thread-extraction technique with great performance potential [71, 88]. PS-DSWP partitions the $\mathrm{DAG}_{\mathrm{SCC}}$ into pipeline stages such that all communication and synchronization flow forward in pipeline order (i.e. forbidding

cyclic communication among worker threads). PS-DSWP delivers scalable speedups when a large *parallel stage* is available; conversely, PS-DSWP does not transform the code when no significant parallel stage is present.

PS-DSWP requires slightly more dependence information than is present in the $\mathrm{DAG}_{\mathrm{SCC}}$. Beyond the $\mathrm{DAG}_{\mathrm{SCC}}$, PS-DSWP classifies each SCC as either *DOALL* or *Sequential* according to the absence or presence of loop-carried dependences. Parallel stages are assembled from the DOALL SCCs such that no loop-carried dependence exists among any instruction assigned to the parallel stage. Algorithm 2 does not guarantee that sufficient queries will be performed to discriminate DOALL and Sequential SCCs. To support PS-DSWP, the algorithm must perform additional queries to classify each SCC as DOALL or Sequential. These additional queries are still fewer than the full PDG and $\mathrm{DAG}_{\mathrm{SCC}}$ guides the compiler to search for such queries. As a further optimization, $\mathrm{DAG}_{\mathrm{SCC}}$ construction may abort early if the $\mathrm{DAG}_{\mathrm{SCC}}$ becomes so constrained that PS-DSWP cannot extract a significant parallel stage.

We extend Algorithm 2 for the needs of PS-DSWP in Algorithm 6. The routine `checkReflexiveLC` checks for loop-carried dependences from any operation in a candidate DOALL SCC to itself, stopping after it finds one. `checkWithinSccLC` checks for loop-carried dependences from any operation located in a candidate DOALL SCC to any other operation in the same SCC. The latter contains the former, but experience suggests that prioritizing reflexive queries tends to exclude many components from the parallel stage after only a linear number of queries, whereas querying in `checkWithinSccLC` is quadratic. At the end, the algorithm invokes `checkWithinSccLC` again since components have grown, potentially including more loop-carried dependences. These checks are cheaper than full PDG construction since they only query among candidate DOALL SCCs, not all SCCs.

In PS-DSWP parallelization, loop-carried dependences between DOALL SCCs prevent the mutual assignment of those components to a parallel stage [71]. The routine

Algorithm 6: PS-DSWP-Aware computeDagScc(V)

```
let E ≔ computeRegisterDeps(V) ∪ computeControlDeps(V);
let TopSort ≔ TarjanSCC(V, E);
abortIfPsInsubstantial(V,E,TopSort);
let E ≔ E ∪ checkReflexiveLC(V);
abortIfPsInsubstantial(V,E,TopSort);
let E ≔ E ∪ checkWithinSccLC(TopSort) ;
abortIfPsInsubstantial(V,E,TopSort);
let E ≔ E ∪ withTheGrain(E, TopSort);
while true do
    let E′ ≔ againstTheGrain(TopSort);
    if E′ = ∅ then
        break;
    end
    let E ≔ E ∪ E′;
    let TopSort ≔ TarjanSCC(V, E);
    abortIfPsInsubstantial(V,E,TopSort);
    let E ≔ E ∪ withTheGrain(E, TopSort) ;
end
let E ≔ E ∪ checkBetweenDoallSccs(TopSort) ;
abortIfPsInsubstantial(V,E,TopSort);
let E ≔ E ∪ checkWithinSccLC(TopSort) ;
return TopSort ;
```

`checkBetweenDoallSccs` performs queries to find such dependences. These checks are cheaper than full PDG construction, since they only consider pairs of DOALL SCCs. `abortIfPsInsubstantial` cancels construction if no substantial parallel stage is present whenever the upper bound on the parallel stage may change. For evaluation, we say a stage is "substantial" if it contains memory accesses or calls.

4.5 Proof of Correctness

We present a proof that our proposed method (Algorithm 2) produces a $\mathrm{DAG}_{\mathrm{SCC}}$ that is equivalent to the one produced by the baseline method (Algorithm 1), both in terms of partitioning the set of vertices V into the same SCCs, and in terms of drawing the same edges between SCCs.

Both algorithms partition the same set of vertices V. Let C_B, C_P represent the components returned by the baseline and proposed algorithms, respectively. Each algorithm computes its own set of edges E_B and E_P, respectively, between pairs of vertices in V. Two components in the $\mathrm{DAG}_{\mathrm{SCC}}$ are connected with an edge if there exists an edge between members of those components: for any components $c_1, c_2 \in C_B$, we write $c_1 \rightarrow_B c_2$ iff there is an edge $\langle v_1, v_2 \rangle \in E_B$ such that $v_1 \in c_1$ and $v_2 \in c_2$. Similarly, for any components $c_1, c_2 \in C_P$, we write $c_1 \rightarrow_P c_2$ iff there is an edge $\langle v_1, v_2 \rangle \in E_P$ such that $v_1 \in c_1$ and $v_2 \in c_2$.

Let $B(v) \in C_B$ denote the strongly connected component which contains vertex v as reported by the baseline algorithm. Let $P(v) \in C_P$ denote the strongly connected component which contains v as reported by the proposed algorithm.

We state our equivalence in Theorems 1 and 2.

Theorem 1 (C_B and C_P induce the same partition of V)**.** *For every* $t, u \in V$, $B(t) = B(u)$ *iff* $P(t) = P(u)$.

Proof. Follows immediately from Lemmas 3 and 5. □

Theorem 2 ($\langle C_B, \rightarrow_B \rangle$ is isomorphic to $\langle C_P, \rightarrow_P \rangle$)**.** *For every* $t, u \in V$*,* $B(t) \rightarrow_B B(u)$ *iff* $P(t) \rightarrow_P P(u)$*.*

Proof. We construct the correspondence $\Psi = B(v) \mapsto P(v)$ for all $v \in V$.

Lemmas 3 and 5 show that Ψ is a bijective function.

Lemmas 4 and 6 show that $t \rightarrow_B u$ iff $\Psi(t) \rightarrow_P \Psi(u)$. □

We prove both Theorems using the following lemmas.

Lemma 1 (Forward Preservation of Edges, Simplified)**.** *Ignoring the* `break` *in Algorithm 4, if* $\langle t, u \rangle \in E_B$ *then* $P(t) \rightarrow_P P(u)$*.*

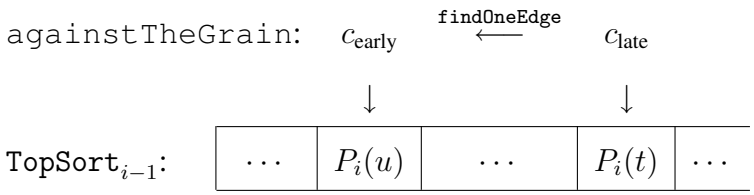
Proof. During an invocation of the proposed method (Algorithm 2), execution will necessarily reach Point Y.

Components evolve during the execution of the proposed algorithm; to avoid confusion we refer to specific versions of the components. Let $P_i(v)$ denote the strongly connected component which contains vertex v at Point Y in the i-th iteration of the loop. In other words, $P_i(v)$ finds the component that contains v within the variable TopSort$_{i-1}$. Note that $P(v)$ is the value of $P_i(v)$ during the final iteration.

We consider three cases based on the relative positions of $P_i(t)$ and $P_i(u)$ in the topological sort of components reported by `TarjanSCC`, observed at Point Y.

Case 1: During any iteration i, $P_i(u)$ appears before $P_i(t)$ in the topological sort.

During that iteration, the invocation of `againstTheGrain` (Algorithm 4) necessarily reaches an iteration during which $c_{\text{late}} = P_i(t)$. It visits every earlier component c_{early}, invoking `findOneEdge` on each until an edge is discovered. Ignoring the `break` statement in Algorithm 4, we will reach an iteration in which $c_{\text{early}} = P_i(u)$.

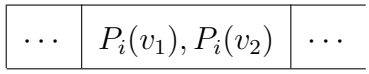


`findOneEdge` (Algorithm 5) will perform queries between the elements of $P_i(t)$ and $P_i(u)$ until an edge is found.

During the execution of the baseline algorithm, the call to Query$(t.\text{inst}, u.\text{inst}, f)$ returns `true` given that $\langle t, u\rangle \in E_B$. Note that `Query` depends only on its arguments, so it behaves the same during the execution of the proposed algorithm.

If `findOneEdge` reaches the iteration where $(v_{\text{src}}, v_{\text{dst}}) = (t, u)$, then Query$(t.\text{inst}, u.\text{inst}, f)$ will again return `true`, thus adding the edge $\langle t, u\rangle$. The only case it may not reach that iteration is when `findOneEdge` finds some other edge between those components. Thus, $P_i(t) \rightarrow_P P_i(u)$.

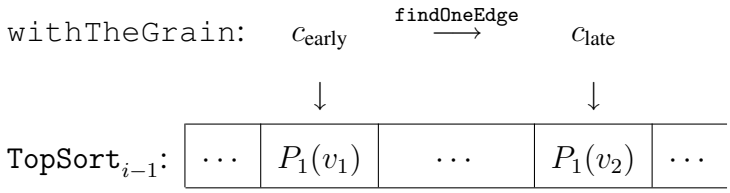
Case 2: During any iteration i, $P_i(t)$ appears at the same position as $P_i(u)$ in the topological sort. That is, $P_i(t) = P_i(u)$.



By reflexivity, $P_i(t) \rightarrow_P P_i(u)$.

Case 3: $P_i(u)$ never appears before or at the same position as $P_i(t)$ in the topological sort during any iteration.

$P_1(t)$ appears before $P_1(u)$ in the topological sort of components during the first iteration of the loop. The topological sort is not updated between Point X and Point Y in the first iteration, so $P_1(t)$ appears before $P_1(u)$ in the topological ordering before the invocation of `withTheGrain` (Point X in Algorithm 2).



The algorithm `withTheGrain` necessarily reaches an iteration during which $c_{\text{early}} = P_1(t)$ and $c_{\text{late}} = P_1(u)$. If there is not already an immediate ordering relationship $P_1(t) \rightarrow_P P_1(u)$, `withTheGrain` passes those components to `findOneEdge`. Since $\langle t, u\rangle \in E_B$,

we know that Query(t.inst, u.inst, f) returned `true`. Thus, `findOneEdge` must find an edge (either $\langle t, u \rangle$ or an earlier one) between these components: $P_1(t) \rightarrow_P P_1(u)$.

In all cases, we have $P_i(t) \rightarrow_P P_i(u)$ for some i.

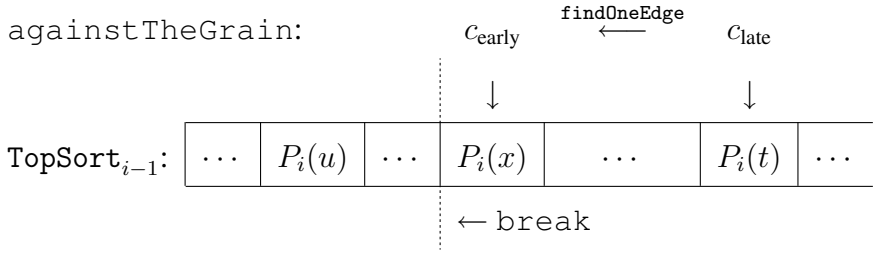
Observe that the proposed algorithm may add edges to the graph, but never removes edges from the graph. Adding edges may cause two separate components to merge into one, but never splits a component. Thus, for any vertex v and iteration i: $P_{i-1}(v) \subseteq P_i(v)$. Since $P(v)$ is the value of $P_j(v)$ in the final iteration j, it follows that $P(t) \rightarrow_P P(u)$. □

Lemma 2. *Considering the* `break` *in Algorithm 4, if* $\langle t, u \rangle \in E_B$ *then* $P(t) \rightarrow_P P(u)$.

Proof. The only difference between the simplified and proposed algorithms occurs in Lemma 1, Case 1: during iteration i, $P_i(u)$ appears before $P_i(t)$ in the topological sort.

The invocation of `againstTheGrain` (Algorithm 4) necessarily reaches an iteration during which $c_{\text{late}} = P_i(t)$. It visits every earlier component c_{early} invoking `findOneEdge` until an edge is discovered.

Suppose there is an intervening component $P_i(x) \neq P_i(u)$ such that `findOneEdge` discovers an edge $\langle w, x \rangle$ from $w \in P_i(t)$ to $x \in P_i(x)$. This edge causes the loop to break before visiting $c_{\text{early}} = P_i(u)$.



After the new edge is found, the algorithm recomputes components and may change their relative positions. Either $P_{i+1}(u)$ precedes $P_{i+1}(t)$ in the topological sort TopSort_{i+1}, or they merge, or $P_{i+1}(t)$ precedes $P_{i+1}(u)$. In the latter case, the subsequent invocation of `withTheGrain` immediately detects an edge from a vertex in $P_{i+1}(t)$ to a vertex in $P_{i+1}(u)$. Thus we need only consider the case in which they maintain their relative topological order.

We argue inductively that such an iteration of Algorithm 2 will be followed by another iteration that falls into Case 1, yet has one fewer intervening component. Assume that $P_{i+1}(u)$ precedes $P_{i+1}(t)$. Observe that the component $P_{i+1}(x)$ cannot appear before $P_{i+1}(t)$ because of the newly discovered edge $\langle w, x \rangle$. Consequently, there is one fewer intervening component that could cause an later invocations of `againstTheGrain` to break. As a new edge $\langle w, x \rangle$ was found, the loop in Algorithm 2 will perform at least one more iteration. Thus, in the next iteration `againstTheGrain` will be one `break` closer to Lemma 1. After sufficient iterations, all intervening components have been eliminated and Lemma 1 Case 1 applies.

□

Lemmas 1 and 2 demonstrate that edges in E_B will order components in C_P. We next strengthen this statement to show that edges between components in C_B will order components in C_P in Lemma 4, but first we prove the following.

Lemma 3 (Wholeness of Components, Forward)**.** *For any vertices $t, u \in V$, if $B(t) = B(u)$ then $P(t) = P(u)$.*

Proof. Vertices t and u belong to the same strongly connected component of C_B, so there is a path from t to u:

$$\langle t, t_1 \rangle, \langle t_1, t_2 \rangle, \ldots, \langle t_{j-1}, t_j \rangle, \langle t_j, u \rangle \in E_B$$

and a path from u to t:

$$\langle u, u_1 \rangle, \langle u_1, u_2 \rangle \ldots, \langle u_{k-1}, u_k \rangle, \langle u_k, t \rangle \in E_B.$$

By Lemma 2 this implies that there is a cycle across the corresponding components of C_P: $P(t) \rightarrow_P P(t_1) \rightarrow_P \ldots \rightarrow_P P(t_j) \rightarrow_P P(u) \rightarrow_P P(u_1) \rightarrow_P \ldots \rightarrow_P P(u_k) \rightarrow_P P(t)$.

This, in turn, implies that $P(t) = P(t_1) = \ldots = P(t_j) = P(u) = P(u_1) = \ldots = P(u_k)$. □

Lemma 4 (Preservation of Structure, Forward)**.** *For any vertices $t, u \in V$, if $B(t) \rightarrow_B B(u)$ then $P(t) \rightarrow_P P(u)$.*

Proof. By definition of $\rightarrow_B$, there is an edge $\langle x, y \rangle \in E_B$ such that $x \in B(t)$ and $y \in B(u)$. By Lemma 2 we know $P(x) \rightarrow_P P(y)$.

Since components are a partition of all vertices, $x \in B(t)$ implies $B(t) = B(x)$. Similarly, $B(u) = B(y)$.

By Lemma 3, $P(t) = P(x)$ and $P(u) = P(y)$.

Thus, $P(t) \rightarrow_P P(u)$. □

Lemma 5 (Wholeness of Components, Reverse)**.** *For any two vertices $t, u \in V$, if $P(t) = P(u)$ then $B(t) = B(u)$.*

Proof. Vertices t and u belong to the same strongly connected component of C_P, so there is a path from t to u:

$$\langle t, t_1 \rangle, \langle t_1, t_2 \rangle, \ldots, \langle t_{j-1}, t_j \rangle, \langle t_j, u \rangle \in E_P$$

and a path from u to t:

$$\langle u, u_1 \rangle, \langle u_1, u_2 \rangle \ldots, \langle u_{k-1}, u_k \rangle, \langle u_k, t \rangle \in E_P.$$

Since the baseline performs *all* queries, $E_P \subseteq E_B$, and the same a cycle connects the corresponding components of C_B: $B(t) \rightarrow_P B(t_1) \rightarrow_P \ldots \rightarrow_P B(t_j) \rightarrow_P B(u) \rightarrow_P B(u_1) \rightarrow_P \ldots \rightarrow_P B(u_k) \rightarrow_P B(t)$.

This, in turn, implies that $B(t) = B(t_1) = \ldots = B(t_j) = B(u) = B(u_1) = \ldots = B(u_k)$. □

Lemma 6 (Preservation of Structure, Reverse)**.** *For any vertices* $t, u \in V$*, if* $P(t) \rightarrow_P P(u)$ *then* $B(t) \rightarrow_B B(u)$.

Proof. By definition of $\rightarrow_P$, there is an edge $e = \langle x, y \rangle \in E_P$ such that $x \in P(t)$ and $y \in P(u)$. Since components are a partition of vertices, $P(x) = P(t)$ and $P(y) = P(u)$. By Lemma 5, it follows that $B(x) = B(t)$ and $B(y) = B(u)$. Since $E_P \subseteq E_B$, $e \in E_B$ and therefore $B(x) \rightarrow_B B(y)$. By substitution we obtain that $B(t) \rightarrow_B B(u)$ as desired. □

4.6 Engineering Considerations

As scopes of optimization grow large, so too must graph representing that scope. Large graphs often present performance challenges since their representations do not fit into a processor's cache and graph access patterns are rarely cache-friendly. This section describes engineering considerations and the design of graph data structures for the PDG and the $\mathrm{DAG}_{\mathrm{SCC}}$.

4.6.1 Compact Representation of the Set of Vertices

A program scope is fixed prior to $\mathrm{DAG}_{\mathrm{SCC}}$ construction. This scope generally comprises a set of N instructions syntactically contained within a program loop. To simplify data structures, we first assign each instruction i a unique integer ID $\texttt{VID}(i)$ in the range $[0, N)$. A sorted array of pointers to `llvm::Instruction` objects represents the mapping between instructions and their IDs. Given an ID, we find the pointer to the `llvm::Instruction` in $O(1)$ time; given a pointer to the `llvm::Instruction` object, we find its ID in $O(\log N)$ time.

Note that this instruction-ID mapping is non-deterministic since the relative ordering of pointers will vary across program runs. This non-determinism cascades through subsequent data structures and ultimately makes the Fast $\mathrm{DAG}_{\mathrm{SCC}}$ Algorithm non-deterministic as

```
struct PartialEdge
{
  // Loop-carried, intra-iteration control dependence
  bool      lc_ctrl      : 1;
  bool      ii_ctrl      : 1;

  // Loop-carried, intra-iteration register data dependences
  bool      lc_reg       : 1;
  bool      ii_reg       : 1;

  // Loop-carried, intra-iteration memory data dependences
  bool      lc_mem       : 1;
  bool      lc_mem_known : 1;
  bool      ii_mem       : 1;
  bool      ii_mem_known : 1;
};
```

Figure 4.4: Eight bits characterize the dependences between two vertices.

well. Alternatively, one could construct a deterministic vertex numbering scheme which would result in a deterministic algorithm. A deterministic vertex numbering scheme is easily derived from a total order on the instructions in the loop: sort the vertices and number them in order. A natural choice of total order on vertices is to order them by their instruction opcodes and operands.

4.6.2 Compact Representation of Edges

Control and register data edges are computed eagerly, however memory data edges are computed lazily. Thus, the edge representation must treat memory dependences as three-valued: *absent edge*, *present edge*, or *edge not yet queried*. Eight bits represent the dependence relationship between a pair of vertices in Figure 4.4.

During the execution of the Fast $\mathrm{DAG}_{\mathrm{SCC}}$ Algorithm, edges may be added to the graph. However, edges will never be removed from the graph.

Recall that dependence graphs are not transitive, reflexive, or symmetric. These properties preclude many simplified graph representations, such as union-find or triangular bit-

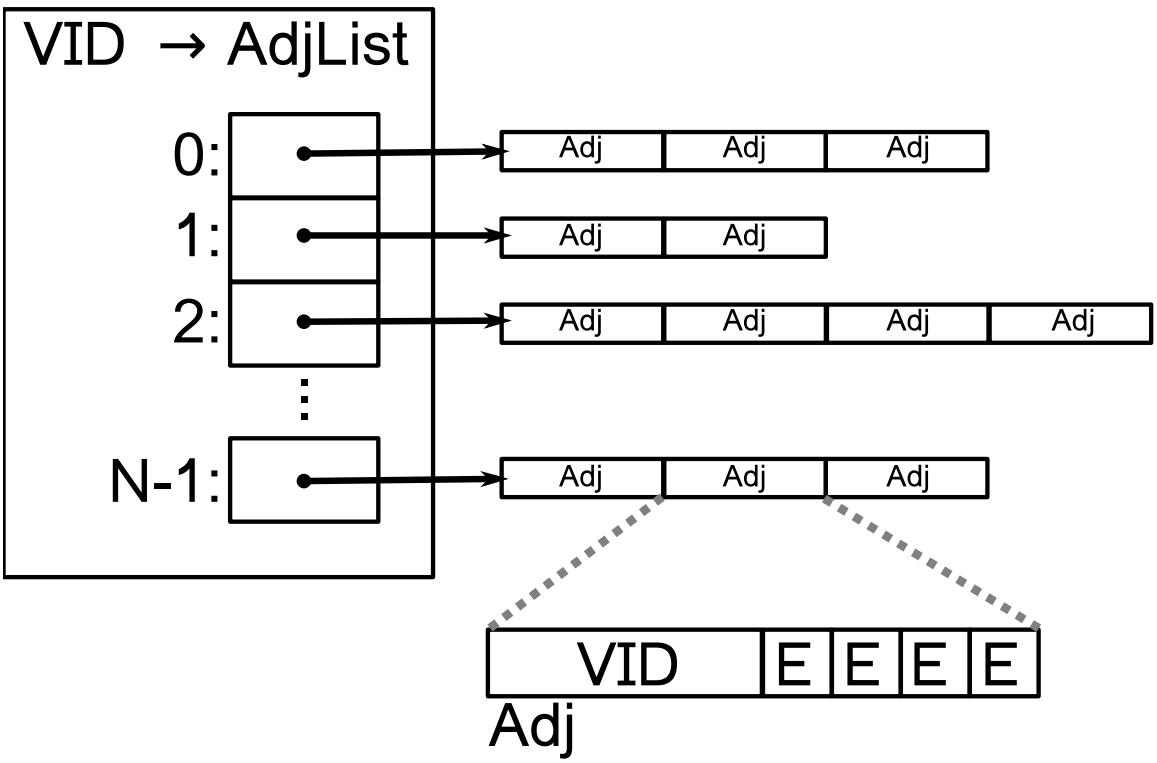


Figure 4.5: A sorted adjacency list representation of the PDG. Each row v encodes all dependences from the instruction i where $\texttt{VID}(i) = v$. Each adjacency object contains a destination VID d and aggregates four 8-bit edges. Those four edges correspond to dependences from instruction i to instructions with VIDs $d + 0$, $d + 1$, $d + 2$, and $d + 3$. Packing several edges into an adjacency object reduces wasted space and improves cache performance.

matrices. Dependence graphs are huge yet sparse. Bit-matrices are wasteful and so large as to incur poor cache performance. Hash-table based graph representations lack spatial locality, and thus are hard on the memory hierarchy.

This work uses a sparse, sorted adjacency list representation for the set of partially known dependence edges. Figure 4.5 illustrates this representation. The graph maintains an array of pointers to adjacency lists, and each adjacency list is a resizable array (a C++ `std::vector`) of adjacency objects sorted by their vertex ID fields. Consider a dependence from an instruction with vertex ID v to an instruction with vertex ID w. To find an edge, we select the adjacency list at row v, and then we binary search within that list for an adjacency object matching w. Overall look-up time is $O(\log n)$. Insertion time is linear in

the size of an adjacency list, but in practice, insertion is efficient because the vast majority of adjacency lists are short and cache performance is good.

Notice that an adjacency object contains four edges rather than one. This is a cache optimization. If, alternatively, each adjacency object contained simply a 32-bit vertex ID and a single 8-bit edge, the adjacency object would require 40 bits yet data alignment rules would force the adjacency object to occupy 64 bits. Consequently, 37.5% of the memory used to represent an adjacency list would be wasted, reducing cache efficiency.

Instead, an adjacency object packs four edges. The adjacency's vertex ID field is constrained to a multiple of four. The look-up operation is updated to binary search the adjacency list for vertex $w - (w \mod 4)$ and then select the $(w \mod 4)$-th edge within the adjacency object. Overall look-up time is still $O(\log n)$ but with improved cache performance and a decreased constant factor.

4.7 Discussion

Performance of the Fast $\mathrm{DAG}_{\mathrm{SCC}}$ Algorithm is presented in Chapter 6. The Fast $\mathrm{DAG}_{\mathrm{SCC}}$ Algorithm compares favorably to the baseline algorithm, which constructs a full PDG and then condenses its components.

4.7.1 Determinism

All proposed algorithms are deterministic, modulo the order in which they iterate over sets of vertices or over components.

In particular, different iteration orders in the four `foreach` loops in Algorithm 5 affects which dependence is discovered first. Similarly, Algorithms 4 and 3 iterate over components in topological order. However, components are only partially ordered, so several topological sorts are possible. The topological sort chosen in these algorithms is precisely the topological sort returned by Tarjan's algorithm, which, in turn, depends upon the order

in which Tarjan's algorithm iterates over vertices and over successors of a vertex.

One can force the algorithm to behave deterministically by choosing a graph representation which supports deterministic iteration orders for all of these cases. The data structures employed here do not offer determinism, specifically the bidirectional map between vertex IDs and pointers to `llvm::Instruction` objects. They could be made deterministic by using a perfect hash function instead of pointer addresses.

4.7.2 Antagonistic Graphs

Tarjan's algorithm to compute the SCCs of a graph is optimal. The performance benefits of the Fast $\mathrm{DAG}_{\mathrm{SCC}}$ Algorithm do not represent an improvement to Tarjan's algorithm. Instead, they stem from an improvement PDG construction time. These improvements are heuristic; there are some *antagonistic* PDGs for which Fast $\mathrm{DAG}_{\mathrm{SCC}}$ will not improve over the baseline or potentially run more slowly.

We will examine two different notions of an antagonistic PDG. The first is constructed to maximize the number of dependence analysis queries performed by the Fast $\mathrm{DAG}_{\mathrm{SCC}}$ Algorithm. The second is constructed to maximize the number of invocations of Tarjan's Algorithm.

Maximum Dependence Analysis Queries

The Fast $\mathrm{DAG}_{\mathrm{SCC}}$ Algorithm's performance improvements over baseline are primarily attributed to a decrease in the number of dependence analysis queries. Thus, one notion of an antagonistic graph is that in which the number of dependence analysis queries approaches that performed by the baseline. Note that our graph representation (see Section 4.6) tracks memory dependence edges as a three-valued quantity; no pair of vertices will be queried more than once, and consequently the worst-case number of memory dependence queries is precisely the number performed by the baseline: n_m^2, where n_m is the number of instructions in the scope which access memory.

To achieve this worst-case quadratic behavior, imagine the degenerate graph which features no register dependences, no control dependences, and no memory dependences. Following Algorithm 2, such a graph will invoke `TarjanSCC` once to compute TopSort_0, invoke `withTheGrain` once to compute E_0, and invoke `againstTheGrain` once during the first iteration of the loop to compute E'. Since the graph contains no edges of any kind, $E' = \emptyset$ and thus the algorithm terminates.

`withTheGrain` issues queries for every pair $\langle c_{early}, c_{late} \rangle$ of components, and `againstTheGrain` issues queries for every pair $\langle c_{late}, c_{early} \rangle$. As there are no edges in this graph, every vertex belongs to its own singleton component in TopSort_0. Hence, these algorithms together will issue queries among every pair of vertices and achieve the worst-case n_m^2 queries.

Maximum Invocations of `TarjanSCC`

The Fast DAG_{SCC} Algorithm uses a topological sort of the partially-constructed graph to identify and cull redundant edges. However, only the final invocation of `TarjanSCC` is returned as a result, and all earlier invocations can be considered overhead of the algorithm. Thus, another notion of the antagonistic graph is one which causes the maximum number of invocations of `TarjanSCC`. Note that the running time of `TarjanSCC` is small in comparison to the cost of dependence analysis queries, and so this notion of antagonistic graph is more theoretical than practical.

Algorithm 2 invokes `TarjanSCC` once at the onset and once every time the call to `againstTheGrain` finds additional dependence edges.

Consider a graph of vertices $v_1, v_2, \ldots, v_n$ which has no register and no control dependences, yet which features a "backbone" of memory dependence edges $\langle v_{i+1}, v_i \rangle$ for all $1 \leq i < n$. In the absence of edges (and thus, absence of cycles), `TarjanSCC` partitions those vertices to singleton components $c_i = \{v_i\}$ and provides an arbitrary[1] order of

[1]Here, *arbitrary* means that there are several valid topological sorts. The actual order is determined by the iteration order from the underlying graph representation. This analysis assumes that `TarjanSCC` and

those components. Assume that order is $c_1, c_2, \ldots, c_n$. Further, assume that `TarjanSCC` delivers a consistent relative ordering of unordered components across every invocation.

Algorithm 2 proceeds as follows. During each iteration $i = 1, 2, \ldots$, the call to `against-TheGrain` discovers the edge $\langle v_{n-i}, v_{n-i-1} \rangle$ and then `TarjanSCC` computes the next topological sort. This continues until all n edges are discovered, thus causing a total of $n + 1$ invocations of `TarjanSCC`.

4.7.3 Integrating Speculation

Speculation is an important part of a parallelization system. Prior work has integrated speculation into a parallelization system by "cutting" speculatively non-existent edges from the PDG and sometimes "de-speculating"—i.e., adding back—those edges when they do not effect the applicability or performance of speculative thread extraction [86]. However, the Fast $\mathrm{DAG}_{\mathrm{SCC}}$ Algorithm is not guaranteed to compute a full PDG data structure. Thus, prior approaches of modifying the PDG to achieve speculation are incompatible with the Fast $\mathrm{DAG}_{\mathrm{SCC}}$ Algorithm. Chapter 5 discusses an elegant means to incorporate speculation into both the CAF and the Fast $\mathrm{DAG}_{\mathrm{SCC}}$ Algorithm.

the graph representation conspire to deliver the particular order that induces this worst-case behavior.

Chapter 5

Speculative Dependence Identification

"When all you have is a hammer,
everything looks like a nail."
—Law of the Instrument,
attributed to either Abraham Kaplan or Abraham Maslow.

Speculation allows optimistic transformation despite the unlikely or spurious dependences which analysis conservatively reports. Through speculation, a thread extraction system makes simplifying assumptions about the program's expected-case behavior to enable an optimistic parallel schedule. Such optimistic schedules may alter observable program behavior in some cases. To preserve behavior in the worst case, the speculative transformation additionally inserts validation code which tests those assumptions at runtime and triggers a recovery routine if those assumptions fail [9, 14, 56, 19, 87]. Instead of incorrect behavior, the speculative application experiences a performance decrease upon misspeculation. Thus, speculation recasts the consideration of *transformation correctness* as the consideration of *transformation performance*.

There are many types of speculation; each asserts different classes of speculative assumptions, requires different validation overheads, and offers different enabling effects. The infrastructure in this dissertation introduces a design pattern that makes speculation

transparent to the optimizing transformation. Consequently, different types of speculation can be selected at compile time. These speculation modules can be combined arbitrarily: their effects compose. The Liberty infrastructure implements several types of speculation.

This chapter explores design constraints on a speculative optimization system and describes the speculation module pattern and how it fits into the greater infrastructure. At a high level, the pattern separates the planning phase—during which no changes are made to the IR—from the action phase. This separation allows the compiler to select the appropriate speculative assumptions before committing to those assumptions via transformation.

5.1 Background

A program dependence graph (PDG) represents a program's statements (or instructions) as vertices and relates those statements via control and data dependence edges (see Chapter 2). Note, however, the distinction between a program's *precise* dependence structure and a PDG data structure. The former is an ideal, whereas the latter is a computationally-feasible, conservative approximation of that ideal. This approximation is sometimes quite conservative, including many dependence edges which have no semblance to real executions of the program.

A speculative optimization system makes simplifying assumptions about the program-under-optimization (PUO) with the goal of enabling a transformation which is not otherwise provably correct. These simplifications preclude certain impossible or unlikely behaviors; this restricted set of admissible program behaviors is mirrored in the PDG as the removal of certain dependence edges. The goal is twofold: to refine the conservative PDG to the program's *precise* dependence structure, and, more aggressively, to refine the PDG to the program's expected case behavior.

Many types of speculation exist. Each assumes a different class of assumptions. Each class of assumptions has different enabling effects on optimizing transformations and in-

curs a different validation cost. No one type of speculation dominates all others. As such, this dissertation proposes composition of speculation modules, so that the speculative optimization system may meet a desired trade-off or so that developers can easily explore new and novel types of speculation.

To plan and then perform a speculative optimization, a compiler juggles the enabling effects and overheads of speculative assumptions in relation to the applicability and performance benefits of the transformation. Additional speculative assumptions may enable increasingly beneficial optimizations. Beyond that, additional assumptions may incur validation overheads that overwhelm the benefit of optimizations. In other words, a speculative optimization system strives to make *enough* simplifying assumptions to enable optimization yet avoid excessive assumptions that may introduce unnecessary overheads. This creates an awkward inter-dependence between planning speculation and planning transformation: the appropriate set of speculative assumptions depends on the desired optimization yet the desired optimization is only possible under certain speculative assumptions.

Prior work "breaks" this inter-dependence with a three-phase scheduling [86]. The compiler first identifies a safe upper-bound on the set of speculative assumptions by pruning those PDG edges which are unlikely to occur. It then plans its transformation with respect to the speculative PDG, resulting in a concrete transformation strategy. Finally, the compiler "de-speculates" the PDG by adding back those speculatively-removed dependences which are not necessary to enable to the concrete strategy. This dissertation introduces a variant of the three-phase approach that does not use the PDG as a primary IR.

5.2 Design Constraints and Design Rationale

Since no type of speculation is perfect, this infrastructure opts to postpone the choice of speculation type. A first design constraint is that each type of speculation can be encoded in a module that is optionally included into the compilation pipeline. These modules should

be, at worst, *loosely coupled* with the rest of the compilation system, i.e., no part of the compilation system should assume that a particular type of speculation is employed. Further, these modules must be structured in such a way that the compiler may employ several speculation modules as easily as it would one, implying the need for composition.

Prior work represents speculative assumptions in the PDG by removing their respective dependence edges, yielding a speculative PDG [86]. Such an approach treats the PDG as a primary intermediate representation (IR) and equates speculative assumptions with the dependences they remove. When considering a suite of several types of speculation (each employing different classes of assumptions), there is a many-to-many relationship between speculative assumptions and dependence edges. Certain dependence edges can be obviated by any of several speculative assumptions; the choice of assumption depends on the relative misspeculation rates and validation costs. Simply removing these edges from the PDG creates an ambiguity that is difficult to reconcile later.

Chapter 4 presented the Fast $\mathrm{DAG_{SCC}}$ Algorithm. This algorithm provides significant reductions in compilation time while maintaining the highest analysis precision. The key insight to these speedups is that the compiler does not need to construct a full PDG in order to construct a $\mathrm{DAG_{SCC}}$. However, if the compiler never constructs the full PDG, it cannot serve as a repository of speculatively removed edges.

This dissertation proposes instead to represent speculative assumptions separately from the PDG, while ensuring that the PDG is consistent with the set of speculative assumptions. Under this model, the PDG is merely an intermediate result used during compilation rather than the primary compiler intermediate representation.

5.3 The Speculation-Module Pattern

Recall from Section 1.6 that the proposed infrastructure consists of a *training phase* that collects profile information, a *planning phase* that analyzes profile data and the program

source to choose a parallelizaton strategy, and a *transformation phase* during which the compiler modifies the program. Implementing a new type of speculation generally requires changes to each of these phases. For example, that type of speculation might require the collection of additional information during profiling, novel interpretation of that information during planning, and emission of validation instructions during the transformation phase. This section presents the speculation-module pattern that modularizes the changes to each phase.

In the proposed system, a speculation module is implemented as four pieces: a manager, a dependence analysis adapter, a validation generator, and a runtime support library. Each piece fits a particular role yet is specialized to the particular type of speculation. The managers and dependence analysis adapters fit into the *planning phase* (see Figure 5.1), the validation generators fit into the *transformation phase* (see Figure 5.2), and the runtime support library is linked against the compiler output.

In essence, the dependence analysis adapters replace dependence identification with *speculative dependence identification*. The optimizing transformation uses dependence information as before, unaware of speculation. Validation generators *patch* the transformation so that it produces correct transformations.

To illustrate this design, the subsequent sections explore each piece in relation to the concrete example of *control speculation*, also known as *biased-branch speculation*.

Control speculation allows the optimization system to ignore unlikely behaviors, such as error handling conditions. Automatic parallelization often speculates that long running loops never take their exit branch, i.e., are infinite. This assumption allows later loop iterations to begin before earlier iterations check their exit condition [12]. One might expect such an assumption to incur huge misspeculation overheads since it must always fail. However, note that this assumption fails only once per loop invocation and hence frequently succeeds in long-running loops.

Under control speculation, certain unlikely targets of conditional branches are assumed

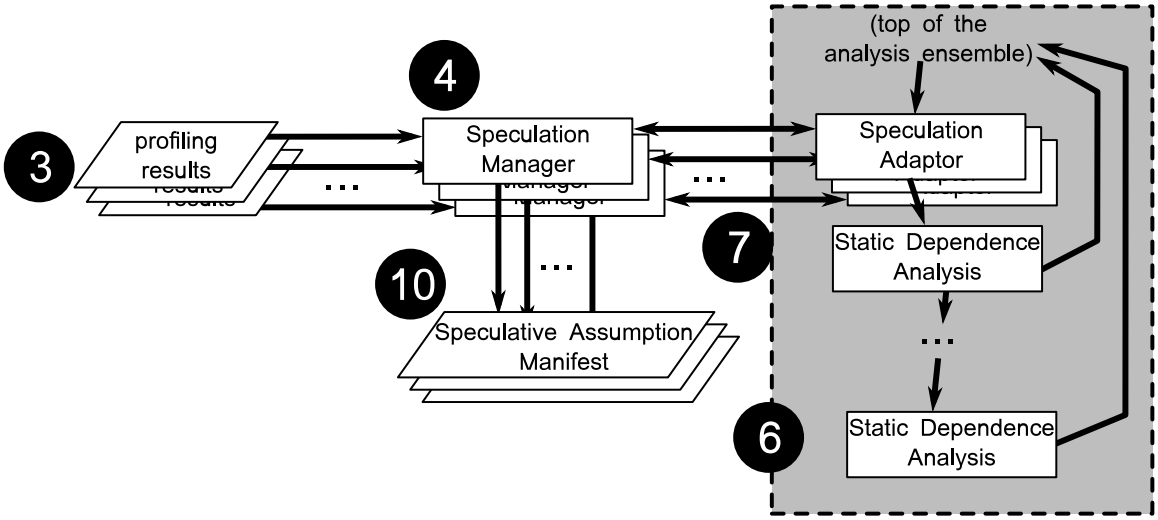


Figure 5.1: A detail of Figure 1.7 showing the roles of speculation managers (4) and analysis adapters (7) in the compiler's planning phase.

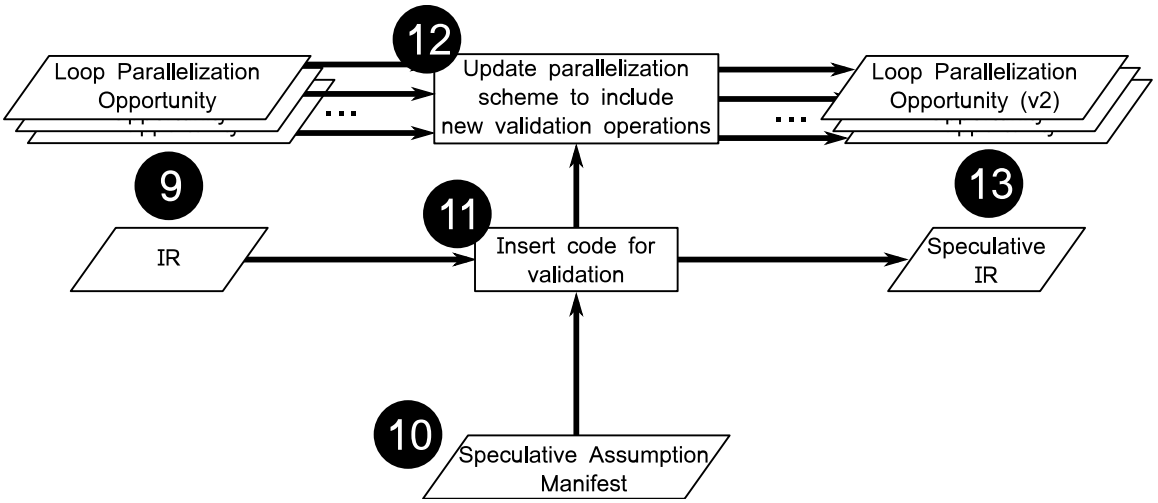


Figure 5.2: A detail of Figure 1.8 showing the role of a validation generator. The generator updates the IR (9) by inserting code (11) which validates all necessary assumptions recorded in the manifest (10) to produce a speculative IR (13). Similarly, generators update the loop parallelization opportunity (12) so that the parallelization transformation distributes these new validation instructions to the appropriate threads.

to be impossible. This assumption simplifies several aspects of the control flow graph. The unlikely target has one fewer predecessor; this reduces the number of feasible incoming values for the Φ-nodes in that block and possibly renders that block unreachable. The speculatively dead instructions in such unreachable blocks can neither source nor sink dependences (see the third condition of Definition 3). Additionally, if the speculated conditional branch has only one feasible successor, it cannot source any control dependences.

5.3.1 Speculation Manager

The speculation manager's role is to determine which speculative assumptions from its class of assumptions can be made with minimal risk of misspeculation (the set of *likely-safe assumptions*), and to track a subset of those assumptions that are necessary to support a specific transformation plan (the set of *necessary assumptions*). These sets are recorded in the *speculative assumption manifest*.

The design pattern does not specify how the speculation manager should determine the set of likely-safe assumptions. Generally, a speculation manager finds these through approximate or probabilistic analysis [14, 19], user annotations to relax program semantics, or by interpreting the behavior observed during a profiling run. All examples in Section 5.5 rely on profiling results to identify high-confidence speculative assumptions.

In control speculation, each speculative assumption takes the form $\langle b, k \rangle$, representing the assumption that a conditional branch b never traverses its k-th successor. To identify likely-safe assumptions for control speculation, A lightweight edge-weight profile measures branch bias on a representative input. The speculation manager interprets the profile to identify branches which exhibit significant bias over sufficient samples. Special care is taken when speculating loop exits: although the compiler may speculate with high confidence that the target loop is infinite, speculating that sub-loops are infinite results in misspeculation during every iteration of the target loop; such frequent misspeculation prevents speedup.

The control speculation manager additionally tracks consequences of its speculative assumptions. After selecting biased branches, it performs a depth-first search over the residual CFG to identify reachable blocks; it records all other blocks as speculatively dead. Additionally, it identifies those blocks which have multiple predecessors in the original CFG yet have a unique predecessor in the residual CFG. The manager records such blocks to speculatively simplify their Φ-nodes.

Runtime validation of control speculation is inexpensive; a program only experiences this validation overhead when a control misspeculation occurs. Consequently, the control speculation manager does not distinguish between the set of necessary assumptions and the set of likely-safe assumptions. For speculation types whose validation costs are higher, the speculation manager is designed to communicate with the analysis adapter to record a more precise subset of necessary assumptions, as described in the next section.

5.3.2 Speculative Dependence Analysis Adapter

A speculative dependence analysis adapter provides an interpretation of likely-safe speculative assumptions (identified by the speculation manager) in terms of dependence analysis. These adapters implement the interface of a dependence analysis (see Section 3.3). Such adapters can be inserted into an ensemble of dependence analysis algorithms, and may collaborate with the other members.

Note that adding speculative adapters to the CAF changes the semantics of all results returned by the CAF and obligates additional bookkeeping. Without an adapter, CAF reports a conservative summary of all possible program executions. Once a speculative adapter is added, the results represent a conservative summary of all possible program executions which *satisfy the speculative assumption*. The speculation manager must track this modified semantics by promoting likely-safe assumptions as necessary assumptions whenever an assumption enables the analysis adapter to report no dependence. Later, the validation generator patches the transformation according to these necessary assumptions in order to

preserve program behavior through speculative transformation. While they answer dependence analysis queries, analysis adapters tell the speculation manager that certain likely-safe assumptions are necessary assumptions, i.e., that the reported dependence analysis results rely upon those assumptions.

The additional bookkeeping obligations are implemented as follows. Whenever the adapter can employ a likely-safe speculative assumption a to report `NoModRef`, it first checks its respective manager to determine whether a is already in the set of necessary assumptions. If not, the adapter may choose either to report a precise speculative result and incur the validation cost of a, or fall back to result provided by the rest of the ensemble (which may include other analysis adapters). This choice is obvious: the adapter should rely on a only if assuming a increases the precision over the rest of the ensemble.

To test whether the assumption increases precision, the adapter *chains* the query (see Section 3.4) to determine whether subsequent analysis implementations can disprove the query without assuming a. Note that this chaining does not increase query latency when compared to an ensemble that does not contain the adapter. If the chained query reports no-dependence, assumption a is unnecessary: the adapter relays the no-dependence result and does not add a to the set of necessary assumptions. Otherwise, the no-dependence result is predicated on a: the adapter instructs the manager to add a to the set of necessary assumptions and returns no-dependence. Thus, the manager accumulates the subset of likely-safe assumptions which affect the reported dependences.

Note that the set of queries processed by speculative adapters is the subset of queries which the Fast $\mathrm{DAG_{SCC}}$ Algorithm identifies as *constructive* (see Section 4.1). Consequently, speculative assumptions are never marked as necessary assumptions unless they disprove dependences that affect the $\mathrm{DAG_{SCC}}$ and in turn affect thread extracting transformations. This restriction, combined with the set of necessary assumptions accumulated in the manager, select the appropriate amount speculation to support a given parallelization.

This pattern of identifying necessary assumptions allows us to prioritize some classes

of speculative assumptions over others. Given two types of speculation S_1 and S_2, it may be that validating assumptions from S_1 is less costly than validating assumptions from S_2. Thus, if there is a dependence which is disproved by either an assumption from S_1 or a different assumption from S_2, the compiler should preferentially rely on the assumption from S_1. For instance, if a certain dependence is disproved using either control speculation or transactional memory, the compiler should use control speculation to reduce overheads. To institute this preference, the compiler developer controls the relative position of speculative adapters in the ensemble (see Section 3.5) to place the adapter for S_1 lower in the ensemble than the adapter for S_2. Thus, S_2 accumulates speculative assumptions only when subsequent analyses (including the adapter for S_1) cannot disprove a dependence.

This criterion for scheduling priority does not conflict with the criterion described in Section 3.5. In general, analysis adapters service queries very quickly because they are merely lightweight abstractions over the speculation managers. A high scheduling priority is consistent with both scheduling criteria.

In terms of the example control speculation, the dependence analysis adapter asserts that several classes of dependences do not manifest. A data dependence from operation t to operation u requires a feasible path of execution that visits t and then u (see third condition of Definition 3). Such a path is impossible if either t or u never executes. Thus, if either is located within a block that the manager identifies as speculatively dead, then no data dependence is possible.

To implement this interpretation, an analysis adapter named `EdgeCountOracle` presents the speculative assumptions through the CAF interface. In particular, it reports `NoModRef` for any `modref_ii` query for which one or both of the instruction operands is located within a speculatively dead basic block.

As noted in the previous section, validation of control speculation is inexpensive. Thus, control speculation's analysis adapter is simpler than the general case: it does not distinguish likely-safe assumptions from necessary assumptions, instead assuming that the

manager reports all assumptions as necessary assumptions.

5.3.3 Validation Generator

After the compiler *planning phase* has determined a plan for optimization (a *loop parallelization opportunity*) and the speculative managers and adapters have identified the set of necessary speculative assumptions (*speculative assumption manifests*), the compiler progresses to the *transformation phase*. Unlike a non-speculative optimization system, a speculative optimization system must additionally insert code that validates speculative assumptions at runtime.

Although inserting validation instructions is generally straightforward, these additional instructions must be considered in relation to the parallelizing transformation. If validation is inserted *before parallelization*, then the parallelizing transformation must know where (i.e. which pipeline stage) to place these additional instructions. If inserted *after*, the validation generator must understand the structure of the code after parallelization, thus strongly coupling speculation to a particular type of transformation. Finally, if validation is inserted by the parallelizing transformation itself, then the parallelizing transformation must be strongly coupled to the types of speculation employed.

To avoid strong coupling, the validation generator takes the approach of inserting validation on the sequential IR, and then updating the loop parallelization opportunity to include these additional instructions. This pattern is very similar to the approach taken by [86]. Specifically, a *parallelization strategy* object represents a partition of the loop's instructions into pipeline stages. After inserting validation instructions into the IR, the validation generator additionally assigns them to stages in the same strategy object.

To apply validation for control speculation to the sequential IR, its validation generator creates new basic blocks along every speculated control flow edge. It inserts code to trigger misspeculation in these blocks: a `call` instruction referencing the procedure `Misspec`, which is implemented in the runtime support library.

To update the loop parallelization opportunity, validation generator must instruct the parallelizing transformation how to treat validation instructions. Validation instructions for control speculation consist of calls to `Misspec` which are placed at the unlikely target of speculated branches. These call instructions are assigned to the pipeline stage which contains the speculated branch instruction.

5.3.4 Runtime Support Library

Each type of speculation includes a runtime support library. At a minimum, this library includes some means to roll back speculative modifications of application state. Several roll back mechanisms are possible. One of the simplest exploits the copy-on-write semantics of the `fork` system call to capture non-speculative state in a separate address space. Alternatively, the runtime support library may maintain a log of reversible memory updates, so that each may be undone upon misspeculation.

Additionally, the runtime support library may contain complicated logic to detect invalid speculative assumption. Because the bulk of this logic can be implemented in the runtime support library, the validation generator may emit simpler validation code, i.e., simple `call` instructions referencing the runtime library.

Control speculation's runtime support library provides only the roll back capability.

5.4 Composability

The Speculation-Module pattern provides clean composability for speculation managers and dependence analysis adapters *for free*. However, achieving composability for the runtime support library (and consequently, in the validation generator) is not automatic. It is possible that two types of speculation require mutually-incompatible modifications to achieve their respective validation. In the worst-case, this may be a fundamental incompatibility, though more often, the incompatibility is the result of few composability hazards

which result from poor design. Careful design achieves composability of the runtime libraries, which in turn provides composability of validation generators.

This section describes a few approaches to refactor runtime libraries for composability: mechanism-not-policy, instrumentation-over-replacement, and idempotency.

5.4.1 Mechanism, not Policy

The first approach favors mechanism over policy. Several types of speculation insert validation instructions for semantically similar yet mechanically different goals. For instance, several types of speculation may feature notions of *transactions*. As a *mechanism* to achieve the semantic notion of a transaction, for instance, validation generators insert `call` instructions referencing support library procedures `begin_tx` and `commit_tx`. Note that the `call` instructions are abstract until the speculatively-optimized program is linked against a concrete runtime library. By deferring the choice of *policy* until link time, this mechanism-not-policy design translates a composability hazard in the validation generators into a composability hazard in the runtime library.

5.4.2 Instrumentation over Replacement

The second approach is to favor instrumentation over replacement. A runtime support library's design prescribes *replacement* if applying speculation entails replacing certain speculated operations with a `call` instruction referencing the runtime library. In contrast, a runtime support library's design prescribes *instrumentation* if applying speculation entails adding `call` instructions referencing the runtime library adjacent to those speculated operations.

For instance, the runtime library and validation generator for control speculation exhibit the instrumentation design because they insert calls to `Misspec` along speculatively dead control flow edges. In contrast, a replacement design would speculatively assert the branch condition before the speculated conditional branch, and then replace the branch with an

unconditional branch. The instrumentation design strives to leave the majority of the IR unchanged while applying speculation, thus introducing fewer hazards to composability.

Separation Speculation (Section 5.5.4) represents a more complicated example of the instrumentation-over-replacement design. Validation of separation speculation requires special allocation routines which force the placement of dynamically allocated objects into specific ranges of the virtual address space. To enforce this, the validation generator replaces the dynamic allocation routine `malloc` with a custom implementation. If done in the obvious manner, this presents a hazard to composability with any other speculation that also wants to replace allocation routines.

Instead, when a validation generator wants to replace an allocator, it replaces it with a parameterized allocator. In particular, separation speculation (or any other validation generator) replaces `call malloc(k)` with `call modified_malloc(k,props)` and initializes the property object `props = call default_properties()`. The parameterized allocator `modified_malloc` is not specific to any particular type of speculation, and is an instance of the mechanism-not-policy design. Each validation generator may then modify the properties object `props` by inserting calls to `set_property` according to the instrumentation-over-replacement design.

5.4.3 Idempotency

The third approach is idempotency. Several types of speculation may, for instance, demarcate transaction boundaries via `call` instructions referencing the runtime library. If the validation generators of each type of speculation redundantly insert these transaction markers, a naïve implementation would attempt to enter nested transactions instead of entering a single transaction. By defining idempotent semantics for such operations, the speculatively optimized program performs the correct behavior. As a further optimization, the compiler may eliminate redundant validation instructions using simple peephole optimizers, though this is unnecessary for correctness.

5.5 Implementations of Speculation

This section briefly describes several types of speculation that are implemented in this speculation module pattern. They are presented here to demonstrate the generality of the speculation module pattern and to encourage discussion of novel types of speculation.

5.5.1 Control Speculation

Control speculation is used extensively by speculative parallelization [12, 40, 45, 86] and is one of the simplest types of speculation. Control speculation is the running example from the previous section.

A control-flow edge counting profiler (LLVM's stock `-insert-edge-profiling`) measures the traversal count of every basic block and control-flow edge under a representative input set. The manager (class `ProfileGuidedControlSpeculation`) interprets this information to estimate branch bias and populates the set of likely-safe assumptions from those branches with significant bias. It calculates a residual control flow graph under those assumptions. Blocks unreachable in the residual CFG are *speculatively dead*. Since validation is essentially free, the speculation manager does not track a set of necessary assumptions, instead assuming it identical to the set of likely-safe assumptions.

The dependence analysis adapter (class `EdgeCountOracle`) asserts the absence of memory dependences to or from any operations located in a speculatively dead basic block.

Following the instrumentation-over-replacement design (Section 5.4.2), the validation generator (class `ApplyControlSpec`) inserts calls to the support library routine `Misspec` at the speculatively-dead targets of conditional branch instructions. This routine signals misspeculation to initiate recovery. The validation generator updates the parallelization strategy so that these new validation instructions are assigned to the same pipeline stage as the speculated branch.

5.5.2 Loop-Invariant Loaded-Value Prediction

Some `load` instructions always read a single, predictable value from memory [24]. Normally, instructions which depend on that predictable value cannot issue until the `load` instruction completes. Value prediction allows such instructions to execute earlier, thus granting the compiler greater freedom of scheduling. Automatic parallelization exploits this increased scheduling freedom for greater concurrency [12, 40, 86]. This section describes a particular type of value prediction [40].

Loop-invariant loaded-value prediction speculates that certain `load` instructions located within loops will read the same value from memory during every iteration of the loop. Under this assumption, such `load` instructions are equivalent to a constant expression and do not require a load from memory. Consequently, these instructions neither source nor sink memory dependences.

A memory profiler (class `liberty::SpecPriv::MallocProfiler`) collects statistics on the set of distinct values read by each `load` instruction located within a loop. Specifically, the profile identifies the subset of `load` instructions whose pointers can be rematerialized in the loop preheader, i.e., loop-invariant pointers. It then observes the values loaded from those pointers in the loop header. The manager (class `ProfileGuidedPredictionSpeculator`) interprets profiling results to identify those `load` instructions that experienced a unique loaded-value over a significant number of observations. These assumptions are tabulated on a per-loop basis, since a `load` may be invariant within one loop, yet variant for a parent loop. The manager populates the set of likely-safe assumptions with the subset of invariant `load`s whose pointer operand can be faithfully rematerialized at the loop preheader. This rematerializability constraint is necessary for proper generation of validation instructions. Similar to control speculation, validating the invariant-loaded-value assumption is inexpensive. Thus, the set of necessary assumptions is identical to the set of likely-safe assumptions.

The dependence analysis adapter (class `PredictionAA`) presents these speculative

assumptions in dependence analysis interface. The adapter responds `NoModRef` to `modref_ii` queries where one or both instruction operands is a `load` instruction which the manager identifies as loop-invariant (i.e., in the set of likely-safe assumptions). Without these dependences, optimization may schedule loop-invariant load instructions earlier.

The validation generator (class `ApplyValuePredSpec`) transforms every loop-invariant `load` instruction in the manager's necessary assumption set, as shown in Figure 5.3. Following the instrumentation-over-replacement design (Section 5.4.2), it rematerializes the `load`'s pointer operand in the loop preheader and reads the assumed-invariant value from it. The validation applicator adds new `store` instructions at the beginning of a loop iteration which enforce this prediction by writing the assumed-invariant value into the storage location. The speculated `load` instruction inside the loop will read the assumed-invariant value. Finally, it inserts a validation test on every loop back edge which compares the assumed-invariant value to the result of loading the value at the end of the iteration. This test ensures that the predicted value is correct during the next iteration, and, inductively, that the predicted value is correct across all loop iterations.

The validation generator must additionally update the parallelization strategy. The rematerialized pointer and the initial load of the assumed-invariant value are outside of the loop and are not recorded in the parallelization strategy. The new `store` instruction in the loop header are assigned to the earliest pipeline stage that contains the speculatively-invariant `load` instruction. The end-of-iteration validation checks are assigned to the latest pipeline that contains the speculatively-invariant `load` instruction.

5.5.3 Memory Flow Speculation

Memory flow speculation assumes the absence of flow dependences between select memory operations. It is quite similar to transactional serializability as speculated by transactional memory systems. This section describes a speculation-module implementation of memory flow speculation which targets the Multi-threaded Transaction (MTX) abstrac-

```
preheader:                          preheader:
  ...                                 ...
                                      ptr_rematerialized = ...
                                      v_invariant = load ptr_rematerialized
  br header                           br header

header:                             header:
  ...                                 ...
  ptr = ...                           // Original load is unused and
  v = load ptr                        // removed via dead-code elimination.
  ...                                 ...
  use(v)                              use( v_invariant )
  ...                                 ...
  br cond, header, exit               br cond, backedge, exit

                                    backedge:
                                      v_test = load ptr_rematerialized
                                      valid = icmp v_invariant, v_test
                                      br valid, header, misspeculate

                                    misspeculate:
                                      call Misspec()
                                      unreachable

exit:                               exit:
  ...                                 ...
```

Figure 5.3: Validation of loop-invariant loaded-values. *(left)* Original IR. The value from instruction `v = load ptr` is invariant with respect to the loop. *(right)* Speculative IR. `ApplyValuePredSpec` rematerializes the pointer `ptr` in the loop preheader `ptr_rematerialized` and loads the assumed invariant value `v_invariant`. It replaces all uses of `v` inside the loop with `v_invariant`. Along the backedge it triggers misspeculation if the assumed invariant value does not match the immediate value.

tion [47, 68, 86], though the implementation is newer [45]. Because memory flow assumptions take a very general form, memory flow speculation provides as much or more an enabling effect than many other types of speculation. However, this generality can lead to expensive validation for certain benchmarks or workloads [13, 45].

Profiling information guides memory flow speculation. A loop-aware memory flow profiler [59] measures the frequency of flow dependences among all memory operations under a representative training input set. A manager (class `liberty::SpecPriv::SmtxManager`) interprets these results. It populates its set of likely-safe assumptions with all pairs of `store` or `load` instructions such that no flow dependence was observed during profiling. The set of necessary assumptions is initially empty.

The dependence adapter responds `NoModRef` to every `modref_ii` query whose operands are assumed independent in the manager's set of likely-safe assumptions. It then promotes those assumptions to the necessary assumption set.

The validation generator (class `liberty::SpecPriv::ApplySmtxSpec`) wraps each loop iteration with instructions to open and close a sub-transaction, following the idempotency (Section 5.4.3) and mechanism-not-policy (Section 5.4.1) designs. The validation generator updates the parallelization strategy so that these transaction boundary calls will appear in each stage of the pipeline. The validation generator also inserts checks on every `store` and select `load`s located in the transaction. `load`s are checked only when they appear within the set of necessary assumptions. Following the instrumentation-over-replacement design (Section 5.4.2), the validation generator applies these checks by inserting calls to `mtx_write` and `mtx_read` before the speculated memory operations. In the parallelization strategy, these new validation instructions are assigned to the same pipeline stage as the memory accesses they guard.

The runtime support library discussed extensively elsewhere [47, 68, 86]. In brief, the runtime library replays all `store` instructions and speculated `load` instructions in a private address space, thus either computing the final, validated version of memory or

detecting misspeculation.

5.5.4 Separation Speculation

Separation speculation partitions a program's allocations into a few "families"[1] of objects and speculatively assumes that every pointer in the program refers exclusively to objects in one family [40]. These families are disjoint; no object belongs to more than one family. Under this assumption, if two pointers reference distinct families, they cannot alias. The pointer-family relation is a coarse approximation of the classical points-to relation, which associates a pointer with the set of objects it may reference. Runtime validation of the pointer-family relation is inexpensive if the number of families is small.

A memory profiler (class `liberty::SpecPriv::MallocProfiler`) tracks every memory allocation and tracks points-to information for every pointer expression in the program. The manager (class `liberty::SpecPriv::Classify`) interprets profiling results to assign each memory allocation to a family. Five families are used, and objects are assigned to families according to secondary criteria discussed in the following sections. The manager then composes each pointer's points-to set with each memory allocation's family assignment to determine the pointer-family relation. The set of likely-safe assumptions contains a pointer-family relation for each loop in the program. Validation is cheap, and so the set of necessary assumptions is identical to the set of likely-safe assumptions.

The dependence adapter (class `liberty::SpecPriv::LocalityAA`) responds to `modref_ii` queries identifying which families the operations will access and reporting `NoModRef` if no family is accessed by both operations. Conversely, if both operations access a common family, the assumed pointer-family relation is insufficient to disprove a dependence among the operations.

Separation speculation uses several tricks to reduce validation costs. At a high level, it modifies allocation so that objects from each family are allocated within a family-specific

[1]The original paper refers to these families and their respective memory arenas as "logical heaps" [40].

region of memory (an *arena*). The runtime system places the arenas at chosen virtual addresses so that the higher-order bits of their virtual addresses carry a family-specific tag value. These tag values survive any well-defined address arithmetic such as array indexing. At runtime, validation instructions use cheap bitwise-arithmetic to extract tags from pointers; one can test whether a pointer references a particular arena in constant time and without requiring any additional bookkeeping or communication among concurrent processes. This scheme of arenas and tagged virtual addresses is similar to segments and segment identifiers used for software fault isolation [89].

To achieve the allocation-family relationship, the validation generator (class `liberty::SpecPriv::ApplySeparationSpec`) replaces all memory allocation and deallocation routines with custom implementations from a runtime library. These implementations allocate objects from the arena corresponding to the desired family. Beyond `malloc` or `alloca`, the validation generator emits code that runs before `main` so that global variables can be reallocated in an arena. Replacing the allocator follows the mechanism-not-policy (Section 5.4.1) and instrumentation-over-replacement (Section 5.4.2) designs.

To validate a pointer-family relationship, the validation generator inserts guard instructions before `load` or `store` instructions. These guard instructions extract a family-tag from the pointer, compare the dynamic tag to the pointer-family assumption, and signal misspeculation upon mismatch. A peephole optimizer proves many of these guards unnecessary or redundant and eliminates them. These checks follow the instrumentation-over-replacement design (Section 5.4.2).

The validation generator also updates the parallelization strategy so that validation instructions will execute in the appropriate pipeline stage after parallelization. Arena-de/allocation instructions are assigned to the same stage as the de/allocation instructions they replace. The per-access guard instructions are assigned to the same stage as the memory accesses they guard.

Separation speculation allows efficient dynamic validation that many spurious memory

dependences do not exist. Additionally, arena memory-management aggregates all objects from one family into a compact, contiguous memory region. This "bonus feature" simplifies a program's memory layout to such an extent that the compiler can statically enumerate a speculatively-complete list of operations which access objects from each family/arena. The next sections describe four types of secondary speculation built on top of separation speculation: read-only speculation, speculative accumulator expansion, speculative privatization, and object-lifetime speculation. These secondary speculation types define the aforementioned inclusion-criteria for separation speculation's families.

Read-only Speculation

In many programs, some memory objects are never modified throughout a scope of optimization. The values of such objects are invariant in that scope, and thus accesses of these objects are independent of all other memory operations in the scope (see condition 2 of Definition 3). Static dependence analysis is sometimes unable to determine this property, so speculating this property is beneficial.

Read-only speculation encodes the assumption that certain objects are invariant throughout the scope. It builds on separation speculation. By aggregating these speculatively read-only objects into a dedicated *read-only family*, separation speculation's default validation ensures that read-only objects are only accessed through the expected pointer expressions.

Ready-only speculation uses the same memory profiler results as separation speculation. The manager (class `liberty::SpecPriv::Classify`) interprets these results to identify speculatively read-only objects. The manager scans the optimization scope to visit every memory update operations. Using the points-to sets (collected during profiling), it identifies the set of objects modified by each update. It marks such objects as *mutable*. Every object not marked mutable is a *read-only* object and is assigned (per separation speculation) to the read-only family.

Recall that separation speculation validates the pointer-family assumption. In this con-

text, the pointer-family assumption establishes that objects in the read-only family are only accessed by those memory operations identified as read-only. By construction, the this set of operations includes no `store` operations. Consequently, speculatively read-only memory operations never experience flow, anti or output dependences.

With this inclusion criterion for the read-only family, the separation assumption implies the read-only assumption. Thus, the manager does not maintain its own assumption sets. Read-only speculation does not need a specific analysis adapter or validation generator. Read-only speculation is strongly-coupled with separation speculation, but introduces no further hazards to composability.

Speculative Accumulator Expansion

Accumulator expansion reorders certain associative and commutative operations so that later loop iterations may execute earlier. For example, the left side of Figure 5.4 shows a loop that is almost parallel except for an accumulator variable `sum`. During each iteration, the value of `sum` is computed using its previous value, resulting in a loop-carried data dependence which serializes iterations. The right side of Figure 5.4 demonstrates a variant of the same program in which the accumulator `sum` is expanded into two accumulators and the final value is resolved after the loop. Expansion forms two independent tasks: the even iterations and the odd iterations.

Accumulator expansion is so important to parallelization that it has earned keyword-status in the OpenMP [2] language extensions. Many automatic parallelization systems rely on accumulator expansion [12, 20, 40, 70, 76, 86].

Generally, the compiler identifies accumulators as values which are repeatedly updated with an associative and commutative operator (a *reduction*) but whose intermediate values are otherwise unused within the loop. Although quite effective, accumulator expansion can be difficult in practice. If the accumulator is located in memory instead of a register temporary, the compiler may fail to prove that *every* access to that storage location is a

```
1  sum = 0;
2  for(i=0; i<N; ++i) {
3    out[i] = work( in[i] );
4
5    sum += ...;
6
7
8  }
9
10 use(sum);
```

```
1  sum_even = 0, sum_odd = 0;
2  for(i=0; i<N; ++i) {
3    out[i] = work( in[i] );
4    if( (i%2)==0 )
5      sum_even += ...;
6    else
7      sum_odd += ...;
8  }
9  sum = sum_even + sum_odd;
10 use(sum);
```

Figure 5.4: Accumulator expansion. *(left)* Each instance of the statement `sum += ...` uses the value of `sum` computed in the previous iteration. This loop-carried data dependence serializes the iterations and inhibits concurrent execution, even though intermediate values of `sum` are otherwise unused. *(right)* Exploiting the associativity and commutativity of the + operator, the accumulator variable `sum` is expanded into two halves which are updated independently. This allows the even iterations and the odd iterations to run concurrently and independently. The final value is resolved after the loop.

reduction or that intermediate values are never used. Speculative accumulator expansion benefits this situation [20, 40, 76]. This section describes a particular notion of speculative accumulator expansion [40].

Speculative accumulator expansion encodes the assumption that certain memory objects are only accessed through reduction operations such as addition, multiplication, maximum or minimum. This assumption is built on top of separation speculation. By aggregating the objects which hold accumulators into a dedicated *reduction family*, separation speculation's default validation ensures that reduction objects are only accessed by reduction operators.

Speculative accumulator expansion uses the same memory profiler results as separation speculation. The manager (class `liberty::SpecPriv::Classify`) interprets these results to identify accumulator objects. The manager scans the optimization scope to visit every `load`-reduce-`store` sequence (i.e., instruction sequences that look like `*p += v`). Using the points-to sets (collected during profiling), it identifies the set of objects updated by these reduction sequences, and marks them as *reduction*. Next, it visits every other

memory accesses in the scope, identifies the objects they access, and remove the reduction marker from those objects if present. Every object marked reduction is a *reduction* object and is assigned (per separation speculation) to the *reduction family*.

Recall that separation speculation validates the pointer-family assumption. In this context, the pointer-family assumption establishes that objects in the reduction family are only accessed by `load`-reduce-`store` sequences and consequently that intermediate values cannot be otherwise observed or modified. Thus, it is safe to reorder the reduction operations so long as the speculative assumptions are valid. With this inclusion criterion for the reduction family, the separation assumption implies the reduction assumption. Thus, the manager does not maintain its own assumption sets.

A dependence adapter (class `liberty::SpecPriv::LocalityAA`) asserts that reduction operations can be reordered by asserting the absence of loop-carried dependences among them. Specifically, it reports `NoModRef` to `modref_ii` queries when either operation is a `load`-reduce-`store` sequence and its pointer-family is the reduction family.

Speculative accumulator expansion does not need a specific validation generator. However, it does require support from the runtime system. In particular, the runtime system must replace the physical page backing of the reduction arena so that each concurrent worker uses an independent copy. Upon entering a parallel invocation, these arenas must be initialized to the reduction operator's identity value (i.e., 0 for $+_{i32}$, $-\infty$ for $\max_{f32}$, etc.). After all workers have finished the parallel invocation, these arenas must be merged after the parallel invocation to deliver the final result.

Speculative accumulator expansion is strongly-coupled with separation speculation. However, it uses features of the virtual memory system to transparently replace the physical backing of the reduction arena; no pointer values change during a parallel invocation. Speculative accumulator expansion does not introduce any additional hazards to composability beyond separation speculation.

Speculative Privatization

In many programs, each iteration of a loop manipulates a data structure, yet that data structure never carries information from one iteration to the next. This reuse does not contribute to the program's semantics but causes anti or output dependences which impose a total order on loop iterations and prevent parallelization. Alternatively, if each iteration uses a *private* copy of that data structure, each iteration accesses its copy independently. Privatization enables parallelization by transforming the code such that each iteration deals with a private copy of the data structure [75, 85]. Privatization of scalars and arrays is so important that it has earned keyword-status in the OpenMP [2] and Parallax [88] language extensions. However, static analysis misses many opportunities for privatization, prompting the development of speculative privatization [20, 40, 76, 93]. This section describes one particular notion of speculative privatization [40].

Speculative privatization encodes the assumption that `load`s from certain *private* objects never read values `store`d during earlier iterations of the loop. To validate this assumption, speculative privatization builds on separation speculation and introduces a *private family* for all such objects. This guarantees, speculatively, that the compiler can statically enumerate a complete list of memory operation that accesses private objects. Further, speculative privatization instruments such *private memory accesses* with validation checks which detect any violations of this privatization criterion.

The manager (class `liberty::SpecPriv::Classify`) uses a loop-aware flow dependence profiler [59] in addition to the separation speculation profiler. It populates its likely-safe assumption set with *private* objects: those objects which lack loop-carried flow dependences. It assigns those objects (per separation speculation) to the *private family*.

A dependence adapter (class `liberty::SpecPriv::LocalityAA`) asserts that private operations are independent from private operations in other loop iterations. Specifically, it reports `NoModRef` to `modref_ii` queries when either operation accesses pointers which reference only the private family.

Following the instrumentation-over-replacement design (Section 5.4.2), the validation generator (class `liberty::SpecPriv::ApplySeparationSpec`) inserts calls to `private_write` and `private_read` before every private `store` or `load` instruction, respectively. These calls allow the runtime library to trace the sequence of stored and loaded addresses to determine whether a privacy violation occurs. These validation instructions are assigned to the same pipeline stages as the memory operations they guard.

The runtime support library implements the privacy check. Additionally, the runtime support library modifies the physical page backing of the private arena such that that each worker process uses a separate physical baking. Additional facilities ensure that the correct final version of private memory is installed after the parallel invocation.

Speculative privatization is strongly-coupled with separation speculation, however it introduces no further hazards to composability.

Object Lifetime Speculation

Object lifetime speculation is inspired by an observed pattern in general purpose codes: some objects allocated within a loop iteration are always deallocated before the end of that same iteration [40, 45]. `load`s from or `store`s to such objects cannot depend on memory accesses in other iterations. However, dependence analysis often fails to identify this case.

Short-lived objects represent a special case of privatization since they cannot carry flow dependences across iterations. The are handled separately from general privatization because validation can be achieved very efficiently for short-lived objects.

Object lifetime speculation encodes assumptions that certain allocation sites are always freed before the end of the iteration. To validate this assumption, object lifetime speculation builds on separation speculation and introduces a *short-lived family* for all such objects.

A memory profiler (class `liberty::SpecPriv::MallocProfiler`) tracks every allocation with respect to loop iterations. Upon traversing a backedge or loop-exit edge of a loop L, the profiler records that all live objects *escape* L. The manager (class

`liberty::SpecPriv::Classify`) interprets profiling results to identify for each loop a set of allocation sites whose objects never escape the loop. The manager populates its set of likely-safe assumptions with these *short-lived object* sets. Validation is cheap, and so the set of necessary assumptions is identical to the set of likely-safe assumptions.

The dependence adapter (class `liberty::SpecPriv::LocalityAA`) responds `NoModRef` to loop-carried queries `modref_ii`$(i_1, \texttt{Before}, i_2, L)$ where instructions i_1 and/or i_2 reference any object that is assumed short-lived.

Beyond the validation provided by separation speculation, object lifetime speculation must validate that short-lived objects never outlive their iteration. The custom allocator and deallocator for the short-lived arena increment and decrement a per-iteration counter, corresponding to the number of live objects in the short-lived arena during that iteration. The validation generator inserts checks at loop back edges which trigger misspeculation if the counter is non-zero. These checks are assigned to the final pipeline stage. The runtime system replaces the short-lived arena's physical page backing so that every worker sees a different arena.

5.5.5 Pointer-Residue Speculation

Separation speculation disambiguates references to different objects, but does not disambiguate references within the same object. Pointer-residue speculation complements separation speculation by working at the sub-object level. It disambiguates different fields within an object and in some cases recognizes different regular strides across an array. Specifically, it characterizes each pointer expression in the program according to the possible values of its four least-significant bits. To the best of my knowledge, pointer-residue speculation has never been published, however, it shares the key insight of address congruence analysis [50].

A memory profiler (class `liberty::SpecPriv::MallocProfiler`) observes the virtual address accessed by memory operations instruction in the program. It tabu-

```
1  struct { int x, y; } pair;
2  char chars[N];
3  int ints[N];
4
5  for(i=0; i<N; ++i) {
6    // Residue set: { 0000_2 }
7    use( pair.first );
8
9    // Residue set: { 0100_2 }
10   use( pair.second );
11
12   // Residue set: all values
13   use( chars[i] );
14
15   // Residue set:
16   // { 0000_2, 0100_2, 1000_2, 1100_2 }
17   use( ints[i] );
18
19   // Residue set: { 0000_2, 1000_2 }
20   use( ints[ 2*rand() ] );
21
22   // Residue sets: { 0100_2, 1100_2 }
23   use( ints[ 2*rand() + 1 ] );
24 }
```

```
// Original IR:
  ...
  // Residue set: { 0100_2, 1100_2 }
  v = load p
  ...

// IR with validation:
  ...
  // bit-vector encodes 4, 12
  mask = 0001000000010000_2
  // extract the residue
  pi = ptrtoint p
  residue = and pi, 0x0f
  // member of residue set?
  element = shl 1, residue
  test = and mask, element
  failure = icmp test,0
  br failure, fail, success

fail:
  call Misspec()
  unreachable

success:
  v = load p
  ...
```

Figure 5.5: *(left)* Different access patterns induce different pointer-residue sets. Disjoint pointer-residue sets contraindicate aliasing. *(right)* Validation of an assumed residue set using bitwise arithmetic.

lates, for each memory access operation, a set of distinct values observed in the four least-significant bits of the pointer. For some access patterns, these sets will contain fewer than sixteen values (see Figure 5.5). Memory alignment constraints ensure that this characterization is consistent across program runs even though virtual addresses of allocations are not. Pointer-residue speculation ensures that every allocation has 16-byte alignment, thus the four least-significant bits of the base address of every object are zero.

The manager (class `liberty::SpecPriv::PtrResidueSpeculationManager`) interprets profiling results. It populates the likely-safe assumption set with pointer-residue

sets if the residue set has fewer than sixteen elements and if the characterization was drawn from sufficiently many observations. The set of necessary assumptions is initially empty.

The dependence adapter (class `liberty::SpecPriv::PtrResidueAA`) receives `modref_ii`(i_1, R, i_2, L) queries and checks whether it can determine pointer-residue sets for operations i_2 and i_2. It then tests whether these residue sets are disjoint with respect to the size of the memory accesses. If the residues are disjoint, the adapter will ultimately report `NoModRef` and install assumed values of the pointer-residue sets into the set of necessary assumptions.

The validation generator (class `liberty::SpecPriv::ApplyPtrResidueSpec`) transforms every memory access operation listed in the manager's set of necessary assumptions to ensure the dynamic pointer values have expected residues. The right side of Figure 5.5 illustrates the new validation instructions. Following the instrumentation-over-replacement design (Section 5.4.2), the new validation instructions consist of bitwise operations to extract a residue and test whether that residue is a member of a statically-encoded bit-vector. These validation instructions are assigned to the same pipeline stage as the speculated memory access.

5.6 Discussion

Speculation augments static dependence identification, but does not replace it. When used sparingly, it eliminates spurious dependences that inhibit optimization. However, the costs speculative validation may prevent net performance gain [13, 45], and so speculation cannot replace static dependence identification.

5.6.1 Speculative Assumptions with Efficient or Scalable Validation

Transactional memory and memory flow speculation require expensive validation. Even though validation can be offloaded to a separate core or itself parallelized, merely logging

the dynamic addresses of every memory access is prohibitive for some workloads [45].

One may reduce validation costs by speculating different properties of the program. Section 5.5 describes many types of speculation whose validation costs are cheaper than memory flow speculation: control speculation, invariant loads, pointer-residue, and separation speculation (including read-only, accumulator expansion, and object-lifetime). These validation techniques are cheaper because they do not introduce additional memory accesses, thread synchronizations, or communications in the common case.

Still, memory flow speculation can remove some dependences which none of these cheaper speculation types can remove. This discrepancy means that memory flow speculation enables more optimizing transformations than all of the cheaper types combined. Further research should investigate additional properties that enable parallelization yet which do not require costly validation.

A different means to reduce validation costs is to reduce the problem size. For instance, speculative privatization (as discussed in Section 5.5.4) is equally expensive as memory flow speculation, but can be cheaper in practice since it validates only memory operations which access objects in the private heap rather than all memory operations in the transaction. Future research should explore hybrids between memory flow speculation and these cheaper speculation types.

5.6.2 Speculation without Profiling

All implementations of speculation in the Liberty Infrastructure (Section 5.5) rely on profiling results to derive high-confidence speculative assumptions. This is not ideal. Speculation imposes several burdens on the developer: the effort to identify a "representative" input set; the time to perform profiling; and, the time to repeat profiling as software evolves. Consequently, reliance on profiling impedes the adoption of speculative optimization.

Some prior work replaces profiling results with probabilistic analysis algorithms [14, 19]. Such algorithms relax the semantics of dependence analysis by imposing a bound

on how frequently an incorrect dependence response would trigger misspeculation. This bound is subtly different than bounding the fraction of incorrect dependence responses, and is difficult to achieve.

Alternatively, some techniques perform an online search for the appropriate speculative assumptions. Adaptive speculation [39] generates multiple versions of the code optimized under varying speculative assumptions, and uses an online adaptation system to switch between these and find the most profitable version.

Chapter 6

Evaluation

"Science without results is just witchcraft."

—First pillar of Aperture Science, *Portal*.

This chapter presents empirical evaluation of the dissertation's techniques. Section 6.1 evaluates the collaborative analysis framework, first in terms of its overall effects on speculative parallelization and then dissecting each design decision. Section 6.2 evaluates the Fast $\mathrm{DAG}_{\mathrm{SCC}}$ algorithm's performance benefit.

6.1 The Collaborative Analysis Framework

This section evaluates the Collaborative Analysis Framework (CAF) described in Chapter 3. First, Section 6.1.1 determines that even *speculative* optimization systems are sensitive to the quality of static dependence analysis and that the CAF is strong enough to turn slowdowns into speedups. Subsequent sections delve deeper. Section 6.1.2 measures absolute analysis precision when compared to a profiling-based "oracle." Section 6.1.2 uses absolute metrics to determine the importance of context-qualifiers in the query language. Section 6.1.2 measures absolute performance with respect to a client of interest: the PS-DSWP thread extraction technique [71] as implemented by the Fast $\mathrm{DAG}_{\mathrm{SCC}}$ Algorithm

(Chapter 4). Section 6.1.2 measures the importance of foreign premise queries and topping to overcome limitations of the best-of-N composition. Section 6.1.3 demonstrates that several analysis algorithms collaborate to disprove multi-logic queries.

6.1.1 Importance of Analysis to Speculative Parallelization

Kim's ASAP System [44] is a speculative automatic thread extraction system that targets clusters. ASAP consists of a compiler and a runtime system. The compiler uses a combination of static analysis and profile-driven speculation to identify and parallelize hot loops from sequential applications. The runtime system provides a transactional memory abstraction to validate speculation and to provide a unified virtual address space among the otherwise disjoint memories available to nodes in the cluster.

Speculation allows ASAP to extract parallelism despite weaknesses of static analysis, but speculation incurs high runtime costs in terms of validation overheads. When more precise analysis is available, ASAP generates more efficient code with lower overheads. ASAP avoids validation overheads completely when non-speculative parallelization is possible.

To analyze the sensitivity of a speculative thread extraction system to dependence analysis precision, I repeated the ASAP evaluation while varying the strength of the dependence analysis subsystem.

To vary analysis strength, I create compiler configurations which employ subsets of all analysis implementations while extracting threads. The total number of distinct configurations is exponential in the number of analysis implementations, and so I selected only 44 configurations in order to estimate the relative importance of each analysis implementation to ASAP's performance on these benchmarks. Specifically, I chose (a) the *full* configuration consisting of all analysis implementations; (b) 21 *leave-one-out* configurations, each consisting of the full set of implementations with one implementation removed; (c) 21 *singleton* configurations, each consisting of a single implementation in isolation; and, (d) the *null* configuration which includes no implementations. We expect (a) to be most precise,

followed by configurations in (b), configurations in (c), and finally configuration (d).

Next, I ran the automatic thread extraction system under each configuration of analysis. Although ASAP could produce 44 variants of each benchmark—one for each analysis configuration—in practice the compiler generated at most two distinct outputs per benchmark.

Seven benchmarks were insensitive analysis precision and four were sensitive. Figure 6.1 summarizes the results of the ASAP analysis-sensitivity experiments for the benchmarks `2mm`, `3mm`, `covariance` and `correlation`. With stronger analysis, ASAP achieves a geomean speedup of 28× on 50 cores, however, when weak analysis is used, those same benchmarks suffer an 11× slowdown. All slowdown measurements are bound by a 24 hour timeout. I was unable to reproduce Kim's results for two benchmarks that crashed during evaluation.

6.1.2 Absolute Precision

We follow Hind's recommendation [31] by evaluating with respect to clients of interest: a *PS-DSWP* [71] client and a Program Dependence Graph (PDG) [21] client.

The PS-DSWP client queries dependence analysis to drive Parallel-Stage Decoupled Software Pipelining thread extraction. PS-DSWP schedules the Strongly Connected Components (SCCs) of the Program Dependence Graph across threads to produce a pipeline execution model. We use a fast algorithm to compute SCCs [41]. Several metrics of PS-DSWP are:

- `%NoDep`: percent of dependence queries for which the ensemble reports no flow, anti, or output dependence. Larger `%NoDep` is consistent with higher precision.

- `NumSCCs`: number of SCCs in the loop's PDG. More SCCs grant PS-DSWP greater scheduling freedom. Imprecise dependence analysis conservatively merges SCCs.

- `NumDoallSCCs`: number of SCCs which lack loop-carried dependences. More is

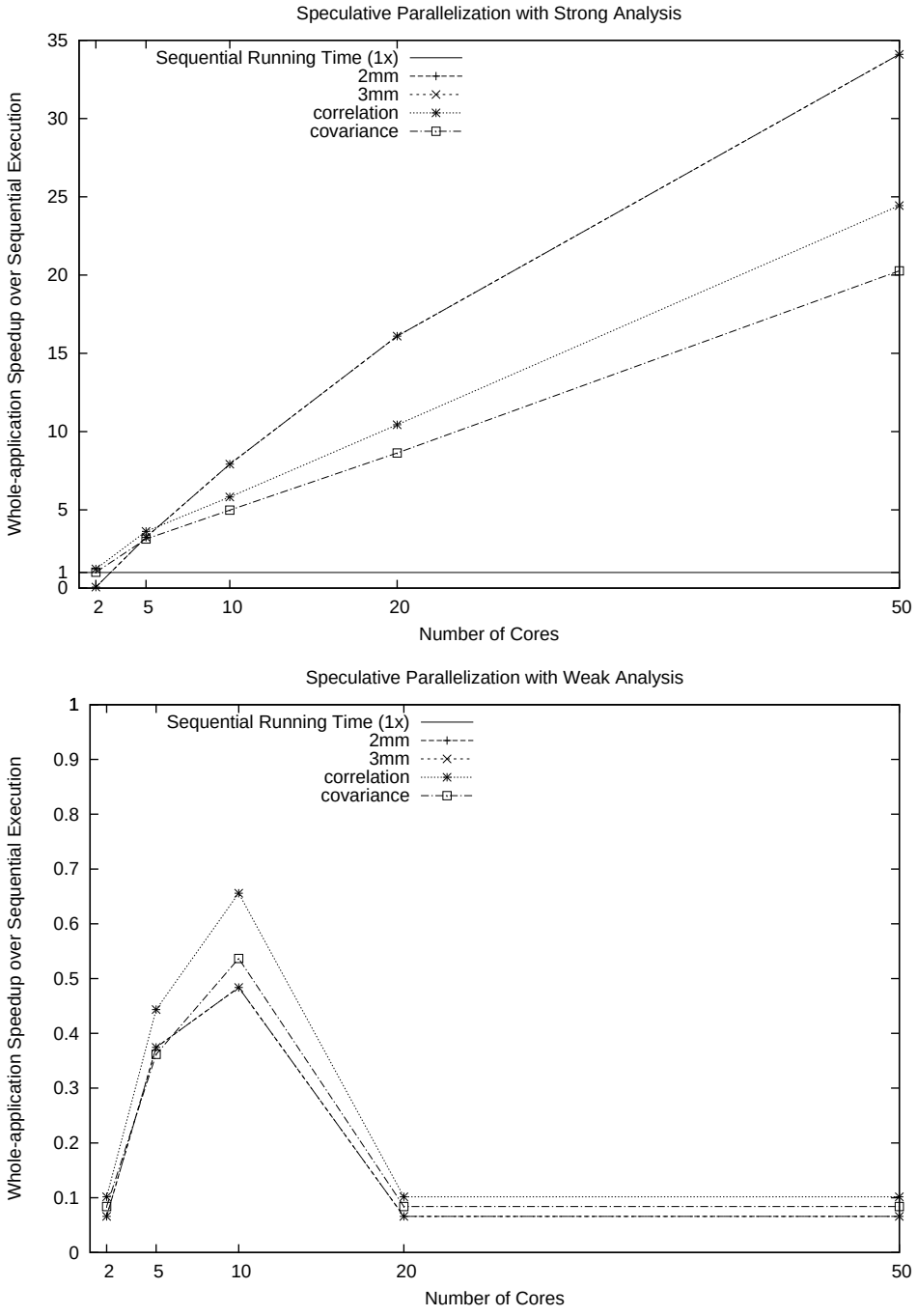


Figure 6.1: ASAP speedup is sensitive to static analysis precision. In four benchmarks, precise analysis allows transformation without speculation or with less speculation, and achieves scalable parallelism. Imprecise analysis forces the compiler to compensate with more speculation on the same benchmarks. Increased validation overheads cause in application slowdown.

better as PS-DSWP schedules DOALL SCCs concurrently.

- `%BailOut`: percent of loops for which the SCC algorithm bails out [41]. Bail-out indicates that PS-DSWP cannot parallelize the code. Smaller is better.

The PDG client performs an intra-iteration and a loop-carried dependence query on each pair of memory operations within each hot loop. The PDG client's `%NoDep` metric is the fraction of queries which the ensemble disproves or reports only a read-after-read dependence. Larger `%NoDep` is better. This metric values every dependence equally.

Both clients are limited to a 30 minute timeout. In the case of a timeout, results indicate progress before a timeout.

We model an oracle from profiling information. We use a loop-sensitive memory flow dependence profiler [59] to identify dependences in applications with its *spec_train* input set. If the profiler observes no memory flow dependence between a pair of operations, the oracle asserts that there is no flow dependence. An analysis-adapter introduces these assertions into the ensemble.

This is not a *true* oracle. Profiles are incomplete because the training input does not induce 100% code coverage. The memory profiler detects only *flow* dependences and cannot assert the absence of anti or output dependences. In these cases, the oracle degrades to static analysis.

In some cases, the oracle is *too* precise because profiling information reflects input-driven program behaviors, whereas static analysis algorithms compute a conservative estimate of program behavior over *all* possible inputs. We do not expect static analysis to achieve oracle results. Despite limitations, this oracle provides a reference for comparison.

Importance of Context-Qualifiers in the Query Language

Recall from Section 3.3 that all CAF queries are qualified against a loop and a temporal relation. These context qualifiers are important for achieving analysis precision by avoiding

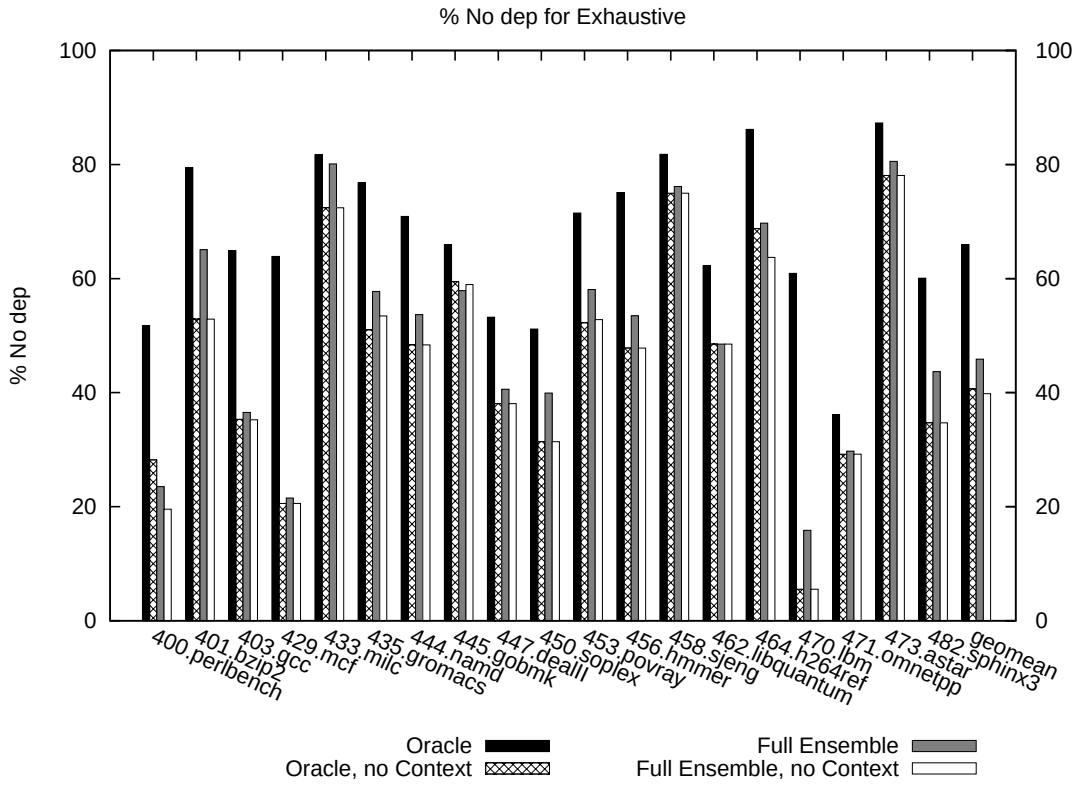


Figure 6.2: Context improves the PDG Client's `%NoDep` metric for both oracle and full ensemble.

unwanted generalization across many paths of execution. To evaluate the effect of context and path qualifiers on precision, we create variants of the PDG and PS-DSWP clients which issue context-blinded queries. We compare the performance of the contextualized and context-blinded variants of the oracle and the full ensemble of static analysis.

Figure 6.2 presents the `%NoDep` metric of the PDG client. When context is removed, oracle performance drops by 25.3% (geomean) and ensemble performance drops by 6.1% (geomean). Figure 6.3 presents results with respect to the `%NoDep` metric of the PS-DSWP client which show a similar decrease in precision. Figure 6.4 shows the fraction of loops within each benchmark for which PS-DSWP bails out (finds no DOALL SCCs). The loss of precision incident on removing context from the queries increases the rate of bail-out by 29.7% for the Oracle and 40.0% for the ensemble, indicating degraded client performance.

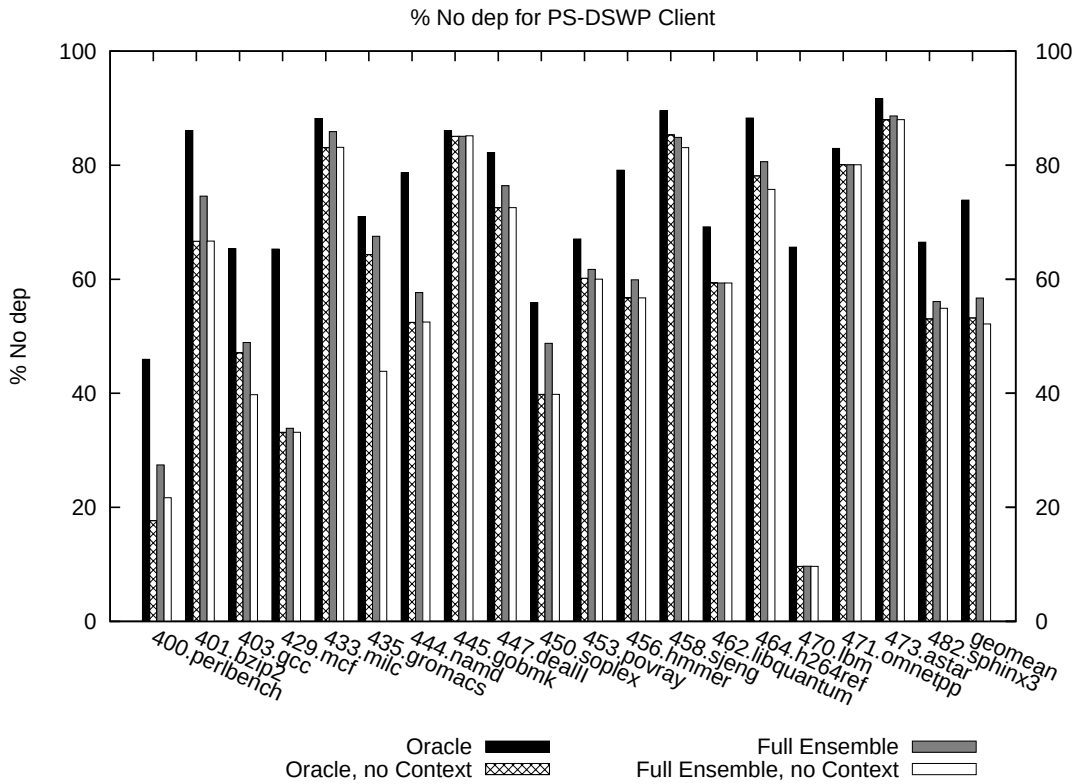


Figure 6.3: Context improves the PS-DSWP Client's `%NoDep` metric for both oracle and full ensemble.

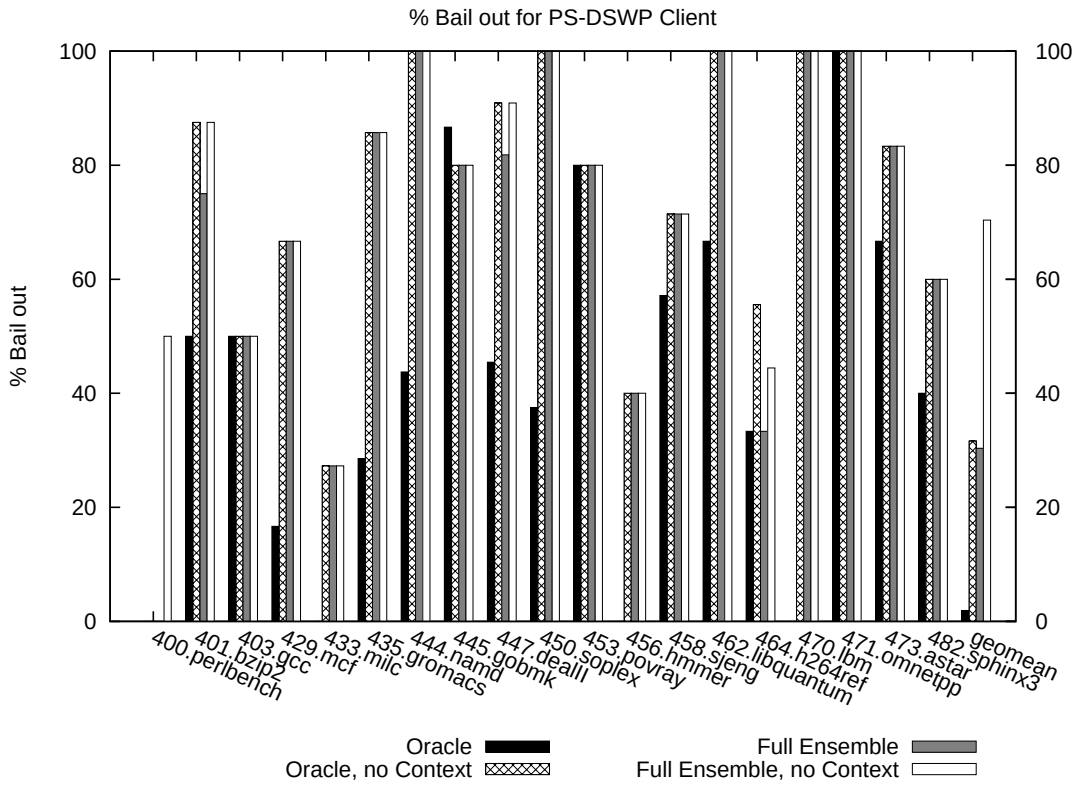


Figure 6.4: Context improves the the PS-DSWP Client's bail-out metric for both oracle and full ensemble.

Precision with respect to PS-DSWP

Of the 169 hot loops found in 20SPEC CPU2006 benchmarks, 20 loops are so constrained by register and control dependences that they have only one SCC. The PS-DSWP client bails-out before it issues any memory query for such loops. Of the 149 hot loops for which PS-DSWP issues queries, the oracle reports worst-case (1 SCC, 0 DOALL SCCs) for 84 hot loops (56.4%). The full ensemble reports the same in these cases.

Among 65 remaining loops, the oracle found more than one SCC. On these loops, the full ensemble reported 27.4% as many SCCs as the oracle (geomean). The full ensemble achieves zero DOALL SCCs for 39 loops. Excluding these loops, the full ensemble reports 66.5% of the DOALL SCCs of the oracle (geomean). Additional analysis algorithms could reduce this difference.

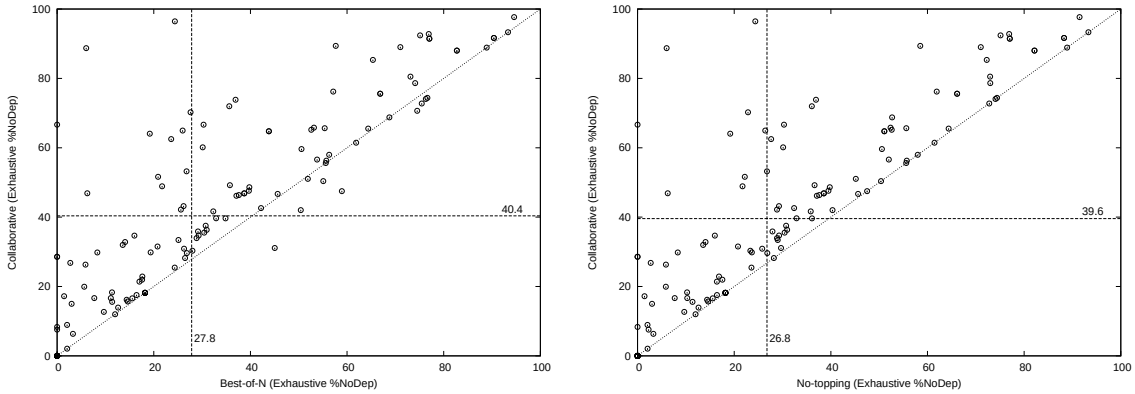
Overall, the PS-DSWP client reports the same number of SCCs for 97 of 169 hot loops, and reports the same number of DOALL SCCs for 98 hot loops, regardless of whether the client employs the oracle or full ensemble.

Chaining and Topping

The proposal encourages development of factored algorithms, arguing that the pattern of chaining and topping achieves precision. To demonstrate the value of chaining and topping, this chapter evaluates alternative means of composition while using the same algorithms.

A naïve alternative to chaining and topping is the *best-of-N* method which passes each query to each algorithm in isolation and returns the most precise answer. This corresponds to the composition pattern used in the LLVM static analysis framework. Figure 6.5a compares the best-of-N method to our proposal. The chaining and topping method performs better than best-of-N on the `%NoDep` metric of the Exhaustive client for all but 10 of 140 loops (timeouts excluded).

Another alternative composition method employs chaining but not topping. To model this method, I modified the framework so that every attempt to top a query instead chains



(a) Collaborative vs. best-of-N composition.

(b) Collaborative vs. no-topping composition.

Figure 6.5: Each point is a hot loop, excluding time-outs and loops without memory accesses. The collaborative approach performs better for loops above the diagonals. Crosshairs show the geometric mean of each dimension, excluding 24 loops with either x=0% or y=0%.

the query, i.e., foreign premise queries are passed only to later members of the ensemble instead of all. Figure 6.5b presents this comparison and demonstrates that chaining and topping combined dominate no-topping.

6.1.3 Collaboration and Orthogonality

One may believe that disparate types of reasoning must be tightly coupled into a single algorithm. This chapter presents results that demonstrate that factored algorithms work together despite strict modularity. These experiments evaluate the marginal benefit of adding new algorithms to the ensemble. Marginal benefit is determined by *orthogonality* and *collaboration*.

A set of analysis algorithms is *orthogonal* if at most one member algorithm disproves any one dependence query. If an algorithm is orthogonal to those in an ensemble, adding it increases the ensemble's precision by the precision of that algorithm in isolation. A set of algorithms is *non-orthogonal* if there is a query which several member algorithms disprove. If an algorithm is not orthogonal to the ensemble, adding it improves precision by an amount less than its precision in isolation. Orthogonality is valued as a software

Orthogonal

	Q1	Q2	Q3	%No
All algorithms	No	No	–	2/3
Algorithm X, isolated	–	No	–	1/3
All algorithms except for X	No	–	–	1/3

Gain $= 1/3 =$ Isolated

Non-orthogonal

	Q1	Q2	Q3	%No
All algorithms	No	–	–	1/3
Algorithm X, isolated	No	–	–	1/3
All algorithms except for X	No	–	–	1/3

Gain $= 0/3 <$ Isolated

Collaboration

	Q1	Q2	Q3	%No
All algorithms	No	No	No	3/3
Algorithm X, isolated	–	No	–	1/3
All algorithms except for X	No	–	–	1/3

Gain $= 2/3 >$ Isolated

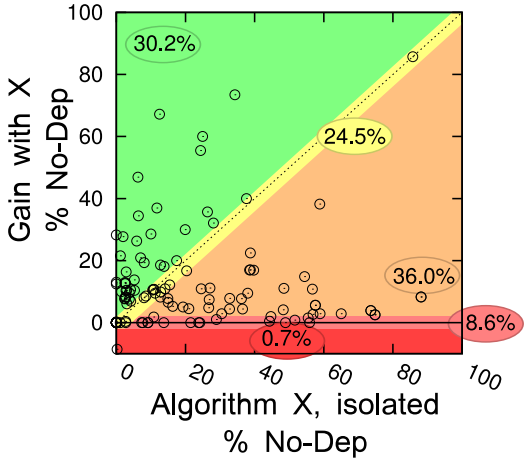


Figure 6.6: Collaboration, orthogonality, non-orthogonality, and anti-collaboration can be measured by differential analysis. For a fixed set of dependence queries, ensembles of different sets of analysis will disprove different subsets of queries.

Analysis Algorithm	Loss vs Isolated Performance (% of Loops)				
	Collab. gain $>$ iso	*Ortho?* gain $=$ iso	*Anti-collab.* gain $< 0 <$ iso	*Non-ortho.* $0 =$ gain $<$ iso	*Anti-collab? Non-ortho?* $0 <$ gain $<$ iso
Kill flow	40.7	44.3	2.1	3.6	9.3
Callsite combinator	32.1	57.1	0.0	0.0	10.7
Array of structures	31.4	50.0	13.6	0.7	4.3
Semi local fun	30.9	50.4	1.4	2.9	14.4
Basic loop	30.2	24.5	0.7	8.6	36.0
Field malloc	21.4	72.1	2.9	2.1	1.4
Unique access paths	13.7	86.3	0.0	0.0	0.0
Auto restrict	10.9	84.1	5.1	0.0	0.0
No capture global	6.4	57.9	3.6	15.7	16.4
Global malloc	1.4	89.9	4.3	2.9	1.4
Phi maze	1.4	56.8	2.9	28.8	10.1
No capture src	0.0	63.3	2.9	20.9	12.9
Disjoint fields	0.0	82.9	5.0	5.0	7.1

Table 6.1: Collaboration, orthogonality, anti-collaboration, and non-orthogonality are observed in the relative strengths of the full, leave-one-out, and isolated ensembles. Each cell is a percentage of hot loops which satisfy an inequality between gain (full - leave-one-out) and isolated ensembles.

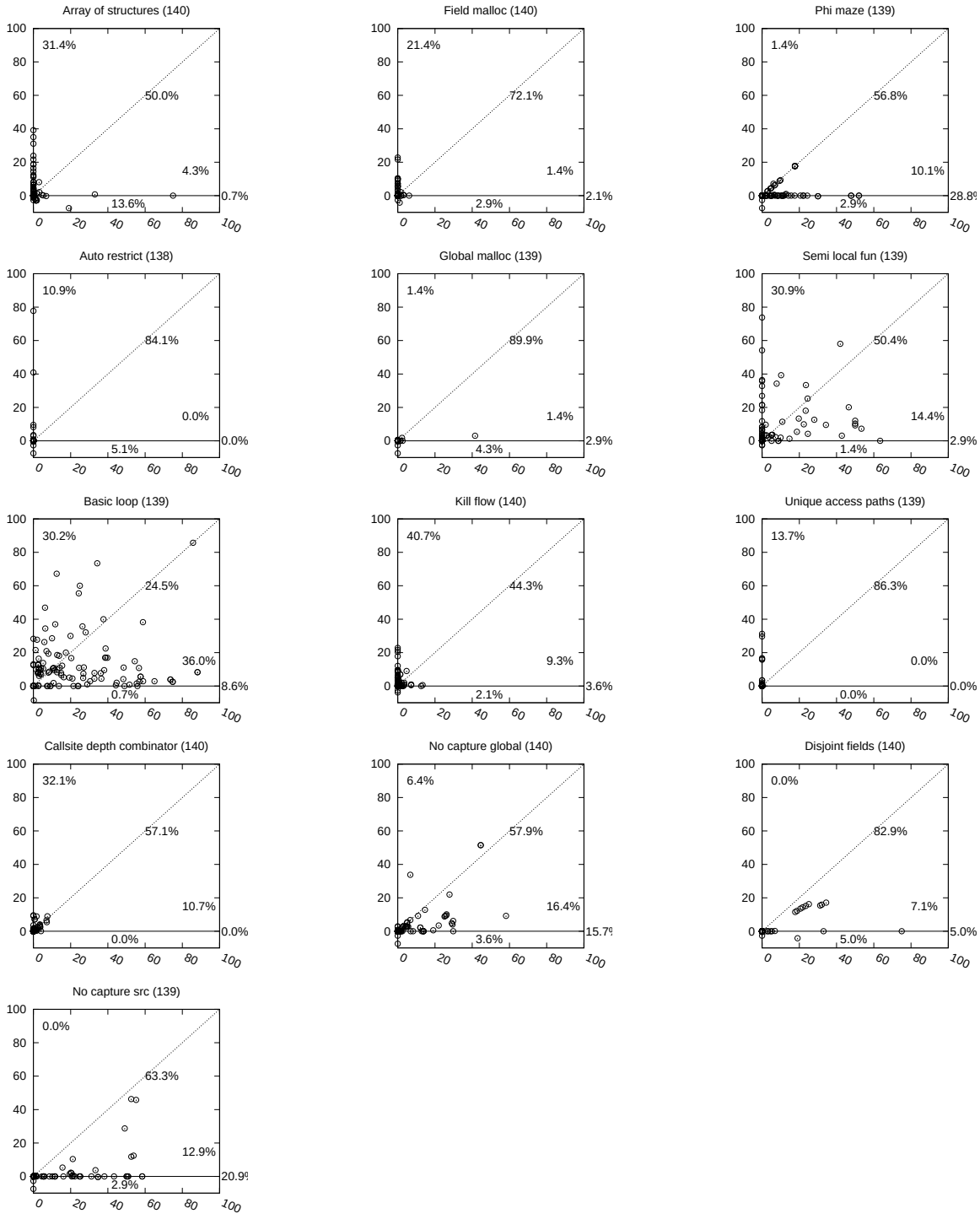


Figure 6.7: These plots show the same data as Table 6.1 in visual form: collaboration per loop of the Exhaustive `%NoDep` metric. The horizontal axis is isolated performance, the vertical axis is gain (full - leave-one-out). Percentages in plot area count loops above, along, or below the diagonal, or along or below the horizontal line. Points above the diagonal indicate collaboration. The parenthetical in each title counts non-timeout loops (total points).

engineering ideal since it is consistent with the minimal amount of software development effort, but non-orthogonality is not detrimental to soundness or precision.

Algorithms in an ensemble *collaborate* if there is a class of dependence queries which the ensemble disproves, yet which no single algorithm disproves in isolation. Conversely, there may be a class of dependences which one algorithm disproves in isolation, but which the ensemble cannot; such cases are *anti-collaborative*. Collaboration is valuable since it indicates a great return on software-engineering effort, and anti-collaboration should be avoided since it indicates a precision bug.

To evaluate collaboration, we compare the Exhaustive Client's `%NoDep` performance metric while varying the ensemble. The *full ensemble* consists of all 13 algorithms. We define *leave-one-out* and *isolated* ensembles for each algorithm A: leave-one-out consists of all algorithms except for A, and isolated consists of A alone. We define gain as the difference between the full ensemble and the leave-one-out ensemble of A. Informally, gain represents the contribution of one algorithm to the ensemble. Orthogonality and collaboration are observed by comparing gain to isolated precision.

Table 6.1 summarizes collaboration and orthogonality experiments by comparing full ensemble, leave-one-out, and isolated performance of each algorithm. Columns present the percentages of loops whose gain is greater than, equal to, or less than its isolated performance. Each loop is an aggregation of queries, so all categories potentially represent a mixture of collaboration, anti-collaboration, orthogonality and non-orthogonality. When gain exceeds isolated, the algorithm contributes more in an ensemble than it does on its own. Such loops are positive evidence of collaboration. Loops whose gain equals isolated performance are consistent with orthogonality. Loops whose gain is zero indicate non-orthogonality. Loops whose gain is less than isolated performance are inconclusive. Loops with negative gains indicate anti-collaboration. Precision bugs in algorithms implementations cause anti-collaboration.

For further detail, we present loop-by-loop data for four select algorithms. The com-

parisons in Table 6.1 correspond to the position of each loop with respect to the diagonals in Figure 6.7. Functor algorithms Array of Structures and Kill Flow show a trend along the isolated=0 border, indicating that these algorithms disprove few queries alone yet help other algorithms to disprove many. Basic Loop demonstrates many loops with isolated>0. However, it also demonstrates collaborative loops where gain>isolated. Although not designed as a functors, Basic Loop collaborates by solving other functors' foreign premises. This also quantifies the degree of anti-collaboration. Array of structures shows anti-collaboration on 13.6% of loops, but most of those are only slightly negative. These cases of anti-collaboration indicate a precision bug that mildy impacts overall ensemble performance.

6.2 The Fast $\mathrm{DAG}_{\mathrm{SCC}}$ Algorithm

To evaluate the Fast $\mathrm{DAG}_{\mathrm{SCC}}$ Algorithm, we implement the baseline (Section 4.2), client-agnostic (Section 4.3), and PS-DSWP-aware (Section 4.4) algorithms in the LLVM infrastructure [51] revision 164307. Each algorithm is augmented with a 30 minute timeout.

Each algorithm uses the same data structures to represent the program dependence graph and strongly connected components. Section 4.6 details the graph data structure. In brief, the PDG data structure is a sorted adjacency-list representation, which performs well since PDGs tend to be sparse graphs. The data structure is capable of representing partial knowledge of memory dependences: between any pair of vertices, a memory dependence is *present*, *absent*, or *unknown*. Thus, none of the algorithms will ever perform the same query more than once. The cost of manipulating the data structure had negligible effect on most experiments.

We evaluated these techniques on 20 SPEC CPU2006 benchmarks [78]. The experiments exclude eight FORTRAN benchmarks because the front-end supports only C and C++. Each benchmark was compiled under two optimization regimens. The less-optimized

	Less-Optimized Regimen					More-Optimized Regimen				
	Hot	Coverage		Size		Hot	Coverage		Size	
Benchmark	Loops	Hottest	Coldest	Largest	Smallest	Loops	Hottest	Coldest	Largest	Smallest
400.perlbench	4	25.6%	5.5%	163 (#1)	9 (#4)	3	25.8%	11.1%	266 (#2)	66 (#3)
401.bzip2	9	73.5%	8.5%	597 (#7)	7 (#9)	9	71.5%	5.6%	2236 (#9)	7 (#8)
403.gcc	16	79.2%	5.0%	5800 (#1)	7 (#12)	11	79.8%	5.4%	11326 (#1)	40 (#2)
429.mcf	7	99.9%	6.3%	81 (#4)	26 (#1)	8	99.7%	8.6%	1352 (#1)	47 (#8)
433.milc	9	52.5%	7.5%	159 (#1)	12 (#9)	15	32.5%	5.4%	298 (#1)	19 (#7)
435.gromacs	5	99.9%	18.6%	671 (#1)	23 (#5)	8	99.4%	6.2%	10191 (#1)	72 (#7)
444.namd	16	99.9%	5.2%	1266 (#10)	9 (#2)	21	100.0%	6.1%	1271 (#14)	66 (#10)
445.gobmk	20	100.0%	5.0%	3868 (#7)	12 (#11)	20	99.9%	5.3%	3099 (#7)	39 (#13)
447.dealII	20	100.0%	5.5%	140 (#17)	10 (#10)	16	100.0%	5.6%	788 (#4)	6 (#6)
450.soplex	6	50.7%	6.4%	118 (#5)	15 (#6)	9	69.4%	5.5%	1034 (#4)	15 (#7)
453.povray	6	99.9%	28.8%	90 (#5)	23 (#6)	7	99.9%	5.6%	258 (#1)	13 (#7)
456.hmmer	6	100.0%	6.4%	277 (#2)	11 (#4)	6	100.0%	7.2%	240 (#1)	13 (#5)
458.sjeng	7	100.0%	9.5%	779 (#4)	147 (#1)	9	99.9%	5.4%	3359 (#7)	13 (#8)
462.libquantum	15	74.2%	4.9%	49 (#4)	5 (#6)	12	94.6%	5.7%	97 (#1)	9 (#11)
464.h264ref	8	100.0%	6.7%	680 (#8)	159 (#3)	8	100.0%	11.1%	1483 (#8)	128 (#3)
470.lbm	2	99.8%	99.1%	475 (#2)	23 (#1)	2	99.6%	99.0%	1175 (#1)	475 (#2)
471.omnetpp	2	100.0%	13.2%	23 (#1)	23 (#1)	2	100.0%	19.0%	37 (#2)	22 (#1)
473.astar	9	65.4%	5.8%	61 (#3)	9 (#9)	12	56.6%	6.7%	238 (#1)	17 (#6)
482.sphinx3	10	95.0%	6.6%	429 (#2)	12 (#10)	8	94.5%	5.0%	2170 (#2)	12 (#1)
483.xalancbmk	2	98.0%	7.2%	28 (#2)	12 (#1)	1	97.6%	97.6%	36 (#1)	36 (#1)

Table 6.2: Hot loops from SPEC CPU2006. "Coverage" is the percent of running time spent in the loop. "Size" is the number of LLVM IR instructions contained in the loop. "Largest" and "smallest" also contain the loop id, where #1 is the hottest loop, and #n is the coldest.

regimen uses `clang -O1`. The more-optimized regimen is designed to create larger scopes that are harder to analyze. Specifically, we apply internalization,[1] devirtualization of indirect calls, and `-O3`.

We profile each benchmark to identify 366 hot loops. Hot loops are those loops whose running time consumes at least 5% of application running time, and which perform at least five iterations per invocation, on average. The hot loops found among the benchmarks are summarized in Table 6.2. It is not always possible to correlate hot loops between the less- and more-optimized regimens; optimization may break a hot loop into several, or reduce the execution time of a loop below the threshold.

Experiments run on an eight core 1.6GHz Xeon E5310. The machine has 8GB RAM and runs 64-bit Linux 2.6.32. All benchmarks are compiled to 64-bit, little-endian code. In this section, we use *instruction* to refer to an LLVM virtual instruction. All measurements

[1]Internalization asserts that the input program is the *whole* program, i.e. that no external libraries reference any of the program's exported symbols. It is similar to marking all global symbols with C's `static` keyword.

experienced negligible variance.

In these experiments, the Fast $\mathrm{DAG}_{\mathrm{SCC}}$ Algorithm use the CAF with an ensemble of nineteen analysis implementations. These analyses are either purely demand-driven or are largely demand-driven, i.e., a significant portion of analysis effort is performed in response to a query, not ahead of time. This configuration of the CAF services most queries quickly: half of all queries take less than 287.6μs (460K cycles); two thirds take less than 601.3μs (962K cycles); 90% take less than 1.0ms (2M cycles). Differences in query running time are due to differences in query complexity: for instance, analyzing a call site is generally more expensive than analyzing a `load` instruction. Across multiple runs, the running time of any one query exhibits negligible variance, suggesting that noise has minimal impact on timing results.

6.2.1 Performance Improvement

The most direct impact of the proposed algorithm is a reduction in $\mathrm{DAG}_{\mathrm{SCC}}$ construction latency.

Figure 6.8a shows the time required to construct a $\mathrm{DAG}_{\mathrm{SCC}}$ for both the client-agnostic and PS-DSWP-aware algorithms. Each point represents a loop from the less- or more-optimized regimen, normalized to the running time of the baseline algorithm (smaller is better). The client-agnostic method is faster for all but 14 of 366 loops.

Performance improvements are due primarily to a reduction in the number of dependence analysis queries. Empirical results concur with the claim that the client-agnostic algorithm normalized running time is linear in the normalized number of queries. The Pearson's Correlation between the normalized construction time and normalized number of queries is 0.63.

Figures 6.8b–6.8d show factors which contribute to the reduction in queries. The fraction of queries performed by the client-agnostic method is related to both the average size of SCCs as well as the number of SCCs, yet is only mildly affected by the size of the region.

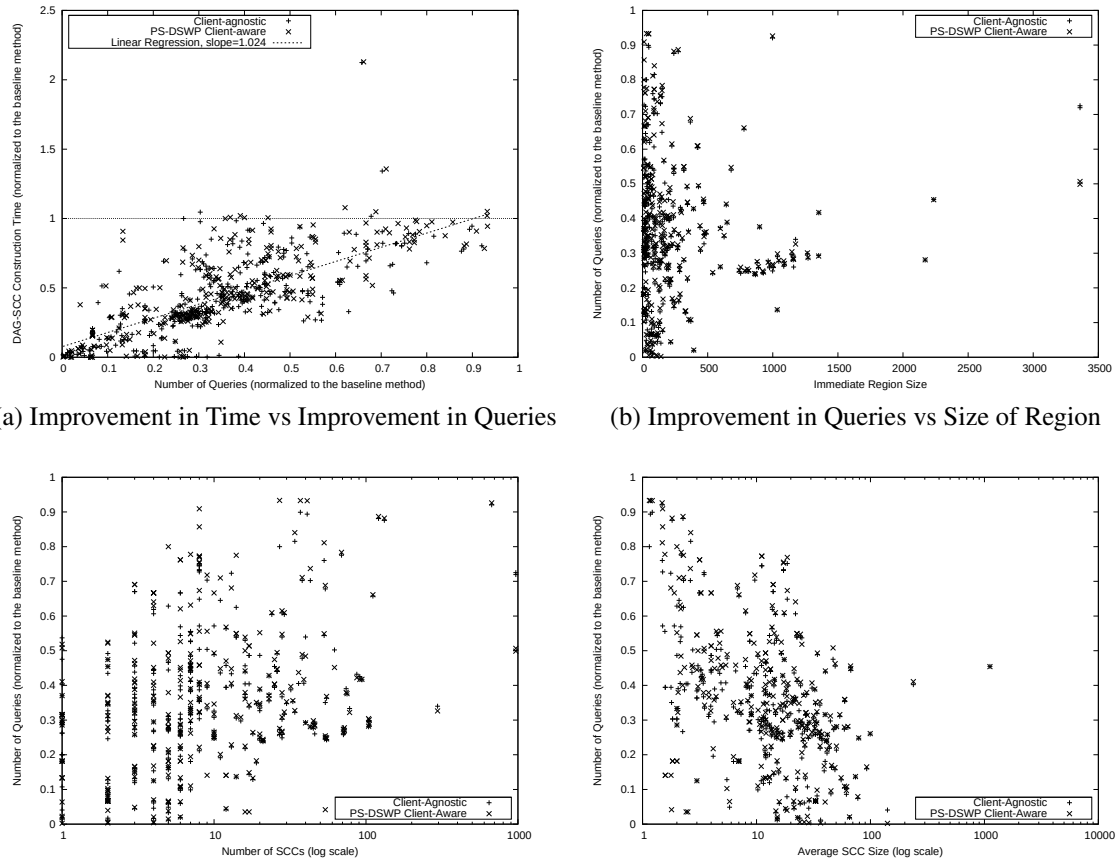
This is because the algorithm elides queries for a class of redundant edges that grows with both average SCC size and number of SCCs (illustrated in Figure 1.6(d)–(e)). Empirical results concur with the claim that the client-agnostic method elides a greater fraction of queries in loops with fewer or larger components. The Spearman's Rank[2] between the average SCC size and normalized number of queries is -0.52. The Spearman's Rank between the number of SCCs and the normalized number of queries is 0.24.

One extreme outlier experiences more than 2× slowdown: the fourth-hottest loop from `458.sjeng`, located in function `std_eval`. In that loop, the proposed methods decrease the number of queries and the time spent on analysis queries. The cost of computing SCCs several times is less than the savings from fewer queries. However, the overhead of manipulating the sparse graph data structure is exceptionally high for this loop, canceling the savings. Further engineering work could reduce this overhead.

Figure 6.9 considers the largest sequences of loops that can be analyzed before varying limits on analysis time (log scale). To simulate a very large application, this experiment allows the compiler to select any of the loops from the entire benchmark suite. The client-agnostic method analyzes more loops than the baseline under the same time constraints. The PS-DSWP client-aware extensions cause a slight performance degradation from client-agnostic yet are still more efficient than the baseline.

Not all loops are equally valuable. Amdahl's law encourages compilers to ration their time budget towards hot loops. Figure 6.10 explores how many hot loops (weighted by coverage) each method analyzes by a certain time. The compiler considers each loop from hottest to least-hot. The vertical axis is the number of loops analyzed, weighted by the relative coverage of each loop. $T_{50\%}$ shows the times when the Baseline, Client-agnostic, and PS-DSWP Client-aware methods reach 50% of cumulative loop coverage or *(30 min)* if they time out first. On average, the client-agnostic method analyzes 50% of cumulative

[2]Spearman's Rank is a measure of statistical dependence [42]. We use Spearman's Rank to support the claim of a *monotone* relationship, which is strictly weaker than a *linear* relationship indicated by Pearson's Correlation.



(a) Improvement in Time vs Improvement in Queries

(b) Improvement in Queries vs Size of Region

(c) Improvement in Queries vs Number of SCCs (log scale)

(d) Improvement in Queries vs Average SCC Size (log scale)

Figure 6.8: Improvement in running time and in number of queries, normalized to the baseline method.

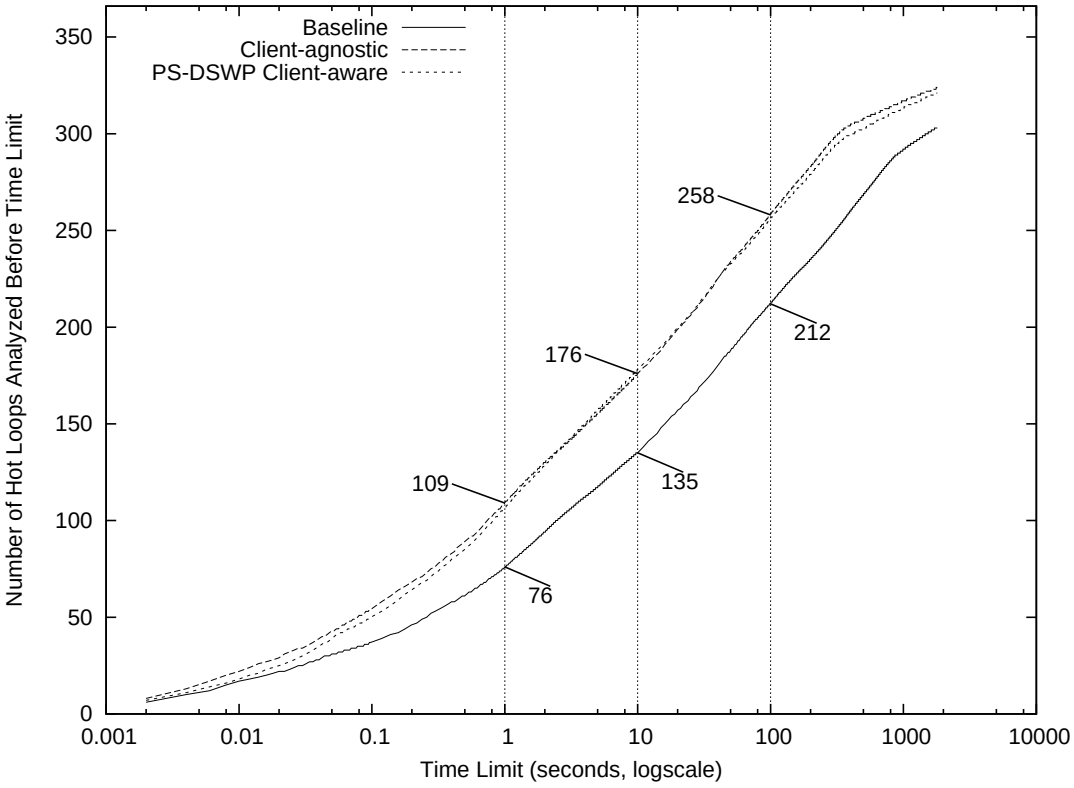


Figure 6.9: Largest sequence of hot loops analyzed before timeout.

loop coverage 111.5s before the baseline, and the client-aware method achieves that 106.1s before the baseline. The proposed methods allow an optimizing compiler to analyze the code which most contributes to running time in shorter use cycles.

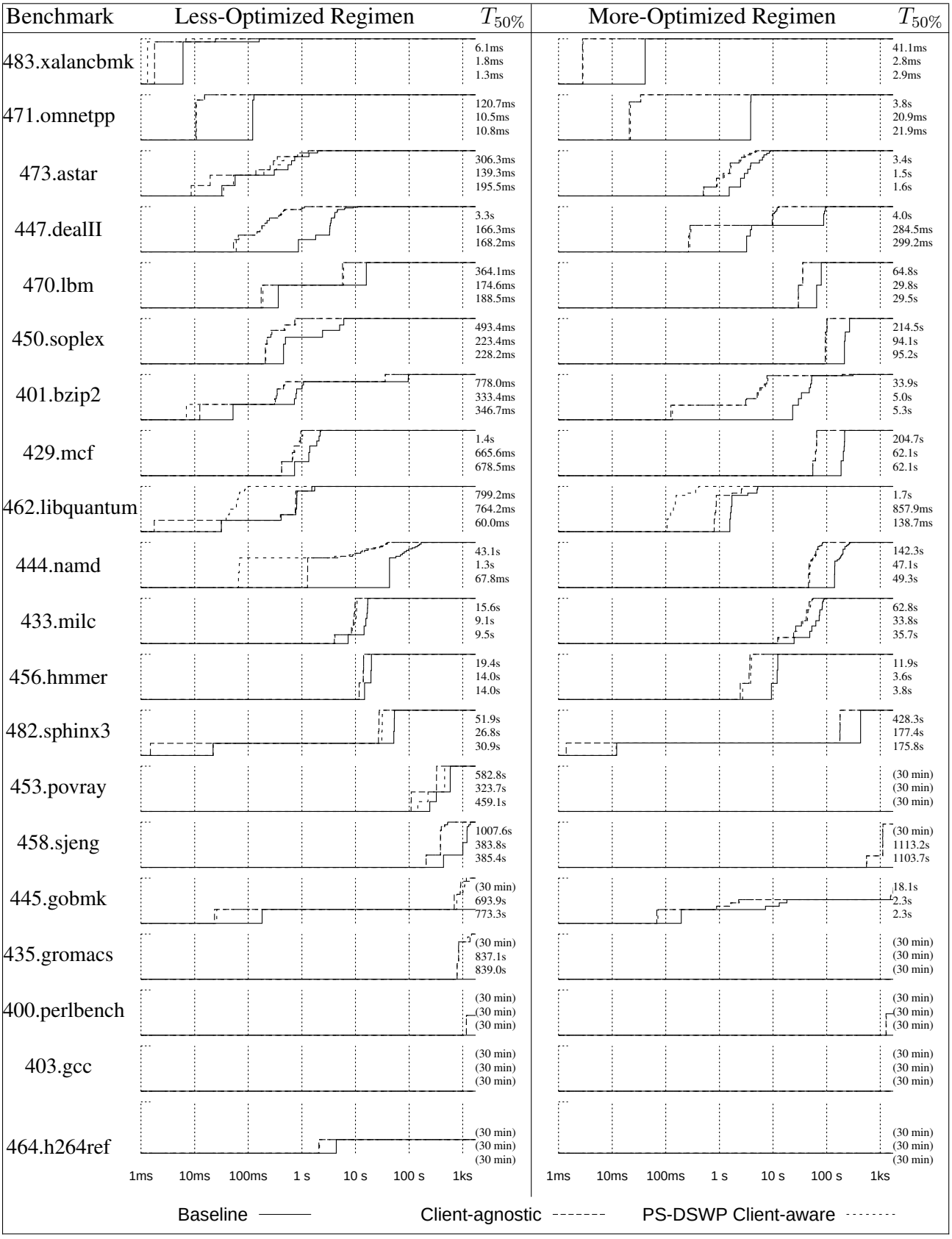


Figure 6.10: The client-agnostic, client-aware and baseline methods analyzing each benchmark. The horizontal axis measures time (s) from 1ms to 30 minutes on a log scale. The vertical axis is the fraction of loops analyzed before a that time, weighted by coverage. The client-agnostic method reaches 50% on average 111.5s earlier than baseline.

Chapter 7

Conclusion and Future Directions

"Science invites; urg'd by the Voice divine,
exert thyself, 'till every Art be thine."
—Mural at the Princeton Post Office.

This dissertation provides a middle-ware for automatic parallelization, including dependence analysis, thread partitioning heuristics, and speculation. Many works have been built upon this infrastructure [38, 65, 45, 69, 36, 37, 57] separately from a paper which cover the infrastructure [40], demonstrating generality and robustness of the components.

7.1 Summary and Conclusions

A central idea of this thesis has been the importance of how a compiler analyzes, manages, and simplifies a program's dependence structure. Certain software architectures are limited, preventing developers from using the appropriate analysis algorithms or speculation types to achieve results in their domain of interest. To overcome these limitations, this dissertation presents the Collaborative Analysis Framework (CAF) in Chapter 3 and the Speculation-Module design pattern in Chapter 5. Another theme has been on compiler scalability as an aspect of usability and adoption. Towards that goal, the CAF allows a

developer to select any subset of dependence analysis algorithms which meet their performance goals, and the Fast $\mathrm{DAG_{SCC}}$ Algorithm in Chapter 4 reduces analysis time regardless of the choice of analysis algorithms.

The CAF allows collaborative composition of varied dependence analysis algorithms. Collaboration allows compiler developers to grow their dependence analysis infrastructure according to the demands of their applications of interest while simplifying the burden of developing each increment. Alternatively, its modularity allows compiler developers to exclude certain analysis algorithms for performance considerations. Concrete implementations (detailed in Appendix A) demonstrate the generality of the framework and the ease with which new implementations are added. Chapter 3 also presents a formalism of the semantics of the query language.

The Fast $\mathrm{DAG_{SCC}}$ Algorithm greatly reduces compilation times by heuristically eliminating unnecessary work during the program analysis phase of compilation. In concert with the CAF (or any demand-driven dependence analysis framework), the Fast $\mathrm{DAG_{SCC}}$ Algorithm identifies certain dependence relationship which cannot affect clients of the $\mathrm{DAG_{SCC}}$ and elides the dependence queries pertaining to those relationships. Chapter 4 also includes a correctness proof.

Speculative dependence identification overcomes the fundamental limitations of static dependence analysis while reducing the overheads of speculative optimization. This hybrid approach offers high transformation applicability and efficient parallel execution. This dissertation presents the speculation-module design pattern, which simplifies design of the optimizing transformation and allows the compiler developer agility in their use of speculation. Compiler developers may mix and match various speculation types, or may rapidly prototype new ones in a general compiler framework.

7.2 Future Directions

This dissertation has not solved the multicore "crisis." Much work remains.

7.2.1 Further Formalization of Dependence Analysis

Chapter 3 presents a formalization of the semantics of the CAF query language. This formalization is an important first step in the formalization of advanced dependence analysis algorithms. This formalization is developed within the ontology defined by the Vellvm project [95, 94], borrowing its definitions for the IR, program and memory states, and its non-deterministic small-step semantics. Although Vellvm is implemented in the Coq proof assistant, the formalization of the query language exists only on paper today.

Mechanizing these formalizations in Coq is a promising future direction. After transcribing the query language's semantics into Coq, various implementations of dependence analysis could be proved correct. Further, collaborative composition itself could be proved correct.

7.2.2 Tools to Aid Development of New Factored Analyses

As noted in Section 3.8.1, the CAF is a work in progress; new analysis algorithms as needed. The process for developing them is driven by observed imprecision. However, tracking this imprecision to a root cause is more difficult than it should be. There is room for tools which automate this process by identifying cases where dependence analysis result are less precise than an oracle.

The challenges then becomes the finding an appropriate oracle and to present the discrepancies in a meaningful manner to the compiler developer. Fortunately, speculative dependence identification serve as an oracle. Every *necessary* speculative assumption (see Section 5.3.1) indicates a potential enhancement to the ensemble of static analysis algorithms.

Presenting enhancement opportunities to the user is more difficult. I envision a tool which maintains a worklist of imprecise dependence queries, and which updates this list in real time as new analysis algorithms disprove them. The tool should also offer some means of regression testing, wherein overly optimistic algorithms can be detected and corrected earlier in the development process.

7.2.3 Efficiently Validated Speculative Assumptions

The cost of speculative validation can overwhelm the benefits of speculative transformation. Much work is necessary to reduce validation overheads. This thesis posits that the best way to reduce these overheads is two fold: using stronger analysis so that fewer speculative assumptions need validation, and choose different classes of speculative assumptions whose validation is more efficient or more scalable.

The dissertation presents several types of speculation that offer reduced validation costs: control speculation, invariant load speculation, read-only speculation, object lifetime speculation, pointer-residue speculation, and speculative accumulator expansion. However, the net enabling effect of all of these still lags that of general memory flow speculation.

More research is necessary into efficiently validated speculative assumptions.

7.2.4 Speculation without Profiling

Profiling imposes significant time and effort burdens on software engineers. Many software engineers do not use profilers or apply profile-guided optimizations. This prevents the adoption of profile-guided speculative transformation. Research on non-profiling means to identify speculative assumptions is necessary.

Appendix A

Analysis Implementations

"All I really want is one sentence
with the word 'conservatively' in the right place."
—Thomas B. Jablin,
Private correspondence, November 2013.

This appendix details the implementations of dependence analysis algorithms used throughout this thesis and presents some high-level themes underlying their operation.

The data dependence relation summarizes the use of storage locations across program executions. Recall from Definition 3 that data dependence between two operations necessitates a common storage location, a feasible path of execution between those operations, and the absence of any killing operations between then which disturbs the flow of data. Dependence analysis algorithms attack one or more of these criteria to disprove dependence.

Sections A.1–A.4 describe some recurring themes employed in dependence arguments. Subsequent sections describe particular dependence analysis algorithms. Sections A.5–A.8 describe the simplest algorithms which address particular features of the IR and the C standard library. Sections A.9 and A.10 present *flow killing* algorithms that address condition 4 of Definition 3. Sections A.11–A.14 describe reachability algorithms, which reason about which object addresses can be stored to particular memory locations. Sections A.15–A.17

describe algorithms which reason about induction variables and the evolution of pointer expressions with respect to loop iterations. Sections A.18–A.22 describe shape analysis algorithms.

A.1 Theme: Lift May-Alias to May-Depend

The first condition of memory dependence (Definition 3) requires that both operations access a common memory location. Consequently, one can *lift* alias analysis algorithms to service dependence analysis queries in some common cases. Specifically, simple `load` or `store` operations depend on one another only if their pointers may-alias. There are so many alias analysis algorithms that this lifting process is common, and so an adapter was created to simplify lifting.

Class `ClassicLoopAA` implements this lifting process. Subclasses of `ClassicLoopAA` override a method `ClassicLoopAA` :: `aliasCheck`$(p_1, s_1, T, p_2, s_2, L)$. Note that `aliasCheck` is distinct from the query method `LoopAA` :: `alias_pp`. The `aliasCheck` is non-recursive: it does not chain queries when it cannot determine a precise answer, relying instead on the `ClassicLoopAA` implementation to perform chaining in the usual fashion.

Upon receiving the query `modref_ii`(i_1, T, i_2, L) `ClassicLoopAA` checks whether both i_1 and i_2 are `load` or `store` instructions. If so, it extracts pointer values p_1, p_2 from those instructions, as well as access size s_1, s_2. It then invokes its method `aliasCheck` $(p_1, s_1, T, p_2, s_2, L)$ to determine an alias; subclasses overload this method to implement an alias test. If these pointers cannot alias, `ClassicLoopAA` reports `NoModRef`. Otherwise, `ClassicLoopAA` chains the original `modref_ii` query.

Similarly, upon receiving the query `modref_ip`(i_1, T, p_2, s_2, L), `ClassicLoopAA` checks whether i_1 is a `load` or `store` instruction. If so, it extracts pointer value p_1 and access size s_1 from i_1. it then invokes its method `aliasCheck`$(p_1, s_1, T, p_2, s_2, L)$ to

determine an alias. If these pointers cannot alias, `ClassicLoopAA` reports `NoModRef`. Otherwise, `ClassicLoopAA` chains the original `modref_ip` query.

The reader may wonder why `ClassicLoopAA` chains the `modref_ii` or `modref_ip` queries instead of chaining an `alias_pp` query. The `modref_*` queries contain slightly more information than `alias_pp` queries by virtue of the instruction operands. These instruction operands not only specify a pointer access, but also a code position where that access occurs. Consequently, by chaining `modref_*` queries, `ClassicLoopAA` preserves an option for subsequent analysis algorithms to exploit that additional information.

A.2 Theme: Conservativism

Human intuition often gives the mistaken impression that certain scenarios are simple. Unless you prove otherwise, no scenario is as simple as it seems.

Rationale: the semantics of the query language generalize all possible executions and all possible inputs. This includes counter-examples that are difficult to imagine.

Consequence: Type annotations have little meaning because some evil instruction performs a reinterpretation cast. Tracing values through memory is difficult because some evil instruction may use or define that memory indirectly through some unknown pointer. Call graphs are inexact because instructions outside of this module call procedures other than `main`. Externally defined procedures can do anything. Procedures will be recursive when you don't want them to be. There are many more.

A.3 Theme: You Cannot Guess an Address

A pointer may only access an object if there is some means for the object address to flow from the allocation of that object. Disproving information flow disproves aliasing.

Rationale: Nothing guarantees that memory allocations occupy consistent addresses across runs of the program. The only algorithm to reliably predict the address returned by

an allocator is the allocator itself; all other algorithms will sometimes guess incorrectly. Any attempt to dereference such an incorrect guess would result in undefined behavior. Dependence analysis may liberally interpret this undefined behavior to its advantage.

Consequence: Two pointers alias only if they are both derived from the same allocation. If the compiler proves that an allocation's address cannot flow beyond some boundary, then pointers derived from values outside of that boundary cannot alias the allocation.

A.4 Theme: Simpler Data Flow on Non-Captured Storage

Similar to virtual registers, memory storage locations hold values. Unlike registers, storage locations can be accessed indirectly through pointers. It is not always possible to enumerate all uses of a storage location, and consequently, it is not always possible to enumerate all definitions and uses of values stored within a given storage location.

Non-captured storage refers to storage locations whose address is never captured. Captures occur when the address is stored into memory by `store` instructions. Captures also occur if the address is passed to an externally defined procedure, since we assume, conservatively, that the procedure conceals capturing `store` instructions. Also, the linker creates opportunities for captures of storage locations whose names are accessible beyond the current unit of compilation; for example, an object file exports all global variables which are not marked private (via C's `static` keyword, C++'s anonymous namespaces, LLVM's `InternalLinkage` linkage type, or related mechanisms).

In the absence of captures, the storage location's address flows only through virtual registers and address computation instructions (see theme *you cannot guess an address* in Section A.3). Using trivial register data flow analysis, an algorithm can enumerate a complete set of `load` or `store` instructions which access a non-captured storage location. For sake of analysis, a non-captured storage location can be treated like a register. Indeed, this is the same reasoning that LLVM's `PromotePass` uses to promote stack allocations

into virtual registers.

Several analysis algorithms simplify information flow through memory by identifying non-captured locations, such as "non-captured fields" (Section A.19) or "unique access paths" (Section A.14).

A.5 Auto-Restrict

- **Implementation:** class `AutoRestrictAA`
- **Tactic:** disprove aliasing; foreign premise queries.
- **Initialization costs:** proportional to size of call graph.
- **Cost / Query:** proportional to number of call sites.
- **Foreign Premise Queries / Query:** proportional to number of call sites.

The "auto-restrict" algorithm simplifies queries among a function's formal parameters when that function's call sites are statically known. Specifically, it formulates foreign premise queries by replacing formal parameter with concrete actual parameter values drawn from each of the function's call sites, in turn.

In essence, this adds C99's `restrict` keyword to formals when appropriate.

A.6 Basic Loop

- **Implementation:** class `BasicLoopAA`
- **Tactic:** disprove aliasing.
- **Initialization costs:** none.
- **Cost / Query:** proportional to the size of a pointer expression's derivation.

- **Foreign Premise Queries / Query:** none.

The "basic loop" algorithm is a straightforward enhancement of LLVM's BasicAA to the CAF interface. It reasons about `null` pointers, as well as the derivation of pointer expressions through cast instructions, Φ-nodes, `select` instructions, and address computations (LLVM's `getelementptr` instruction). It asserts that stack allocations do not alias global variables.

A.7 Φ-maze

- **Implementation:** class `PHIMazeAA`
- **Tactic:** disprove aliasing.
- **Initialization costs:** none.
- **Cost / Query:** linear in the number of instructions in a function.
- **Foreign Premise Queries / Query:** none.

The "Φ-maze" algorithm traces the definitions of pointers through Φ-nodes, pointer casts, and address computations (LLVM's `getelementptr` instruction) to identify a set of allocation instructions. It reports no-alias between pointers when the source-sets corresponding to both pointers are complete and it can demonstrate all sources are disjoint. The Φ-maze algorithm reasons that allocations (stack or heap) are disjoint from one another, from formal parameters, and from global variables. It cannot, however, disambiguate dynamic instances of a single allocation.

A.8 Pure and Semi-Local Functions

- **Implementation:** classes `PureFunAA` and `SemiLocalFunAA`

- **Tactic:** knowledge of library functions.
- **Initialization costs:** linear in height of call graph; linear in size of program.
- **Cost / Query:** linear in number of callsite arguments.
- **Foreign Premise Queries / Query:** linear in number of callsite arguments.

The "pure functions" and "semi-local functions" algorithms employ knowledge of common functions from the C, C++, and POSIX standards. They codify knowledge that certain functions are *pure* (i.e. their semantics do not require a memory access or an observable side effect), that certain functions are *local* (i.e. they may read or write memory referenced by their actual parameters or by a closed set of global variables), or *semi-local* (i.e. local functions which may perform side effects). For example, the function `sin` is pure, the function `memset` is local, and the function `puts` is semi-local. Additionally, these algorithms annotate that certain pointer arguments are read though never written.

At initialization, these algorithms traverse the call graph from leaves to root. By considering all operations within a procedure or its callees, these algorithms infer the pure, local, and semi-local properties of user-code.

Classes `PureFunAA` and `SemiLocalFunAA` respond to `modref_ii` queries where one or both operands is a callsite to a pure, local, or semi-local function. They report `NoModRef` for calls to pure functions. It models calls to local functions as `load`s and `store`s to the callsite's actual parameters, issuing foreign premise queries for each case. It models calls to semi-local functions similarly to local functions, but additionally reports an order between semi-local functions and any other side-effecting operation.

A.9 Kill Flow

- **Implementation:** class `KillFlow`
- **Tactic:** flow killing; foreign premise queries.

- **Initialization costs:** none.
- **Cost / Query:** linear in the number of instructions in any function times the height of the call graph.
- **Foreign Premise Queries / Query:** linear in the number of instructions in an function times the height of the call graph.

The "kill flow" algorithm searches for killing operations along all feasible paths between *source* and *destination* operations. Since there may be infinitely many paths, it restricts its search to blocks which post-dominate the source and dominate the destination. This is a conservative approximation of those paths: these blocks appear on all feasible paths between source and destination.

Among those blocks, it visits every intermediate `store` instruction. It issues foreign premise queries to determine if the intermediate `store`'s pointer *must*-alias with the pointers accessed by the source or destination operations. If so, the kill flow algorithm reports `NoModRef`.

Additionally, the kill flow algorithm may discover call sites between the source and destination operations. These call sites may conceal flow killing operations, so the kill flow algorithm searches inside the callee procedure as well. In particular, it performs a flow killing search among those basic blocks which post-dominate the callee's entry block. This is a conservative approximation of all paths through the function.

Worst-case running time is high. To reduce expected-case running time, the kill flow algorithm caches intermediate facts, for instance, that a given basic block kills a certain set of pointers. Further, it searches intermediate basic blocks according to a breadth-first traversal of the post-/dominator trees, hoping that this will improve performance of its killing cache.

A.10 Callsite Depth-Combinator

- **Implementation:** class `CallsiteDepthCombinatorAA`
- **Tactic:** callsite expansion; flow killing; foreign premise queries.
- **Initialization costs:** none.
- **Cost / Query:** quadratic in the height of the call graph; quadratic in the size of a function.
- **Foreign Premise Queries / Query:** quadratic in the height of the call graph; quadratic in the size of a function.

The "callsite depth-combinator" (CDC) algorithm codifies that the memory behavior of a call site is the sum of the memory behaviors of operations within the callee procedure.

Upon receiving a query `modref`(i_1, T, i_2, L) where either i_1 or i_2 represents a procedure call, class `CallsiteDepthCombinatorAA` expands the query into multiple foreign premise queries—one for each discovered memory operation. It responds to the query with the conservative join of the responses from each foreign premise query, and bails out early if the worst-case answer is achieved.

To perform the expansion, class `CallsiteDepthCombinatorAA` uses subclasses of `InstSearch`. Such iterators traverse the call graph *lazily* and track calling-context for all returned instructions. Calling context allows the iterators to perform extensive flow killing checks (per the kill flow algorithm in Section A.9) and exclude all but the downward-exposed `stores` or upward-exposed `loads`. Further, calling context allows the iterators to substitute actual parameters for formal parameters.

Figure A.1 illustrates the iteration pattern with an example. Subclass `ReverseStoreSearch` finds downward-exposed `store` instructions in callsites, starting from the roots of the post-dominator tree and visiting "later" instructions before the instructions they post-dominate. Subclass `ForwardLoadSearch` finds upward-exposed `load`

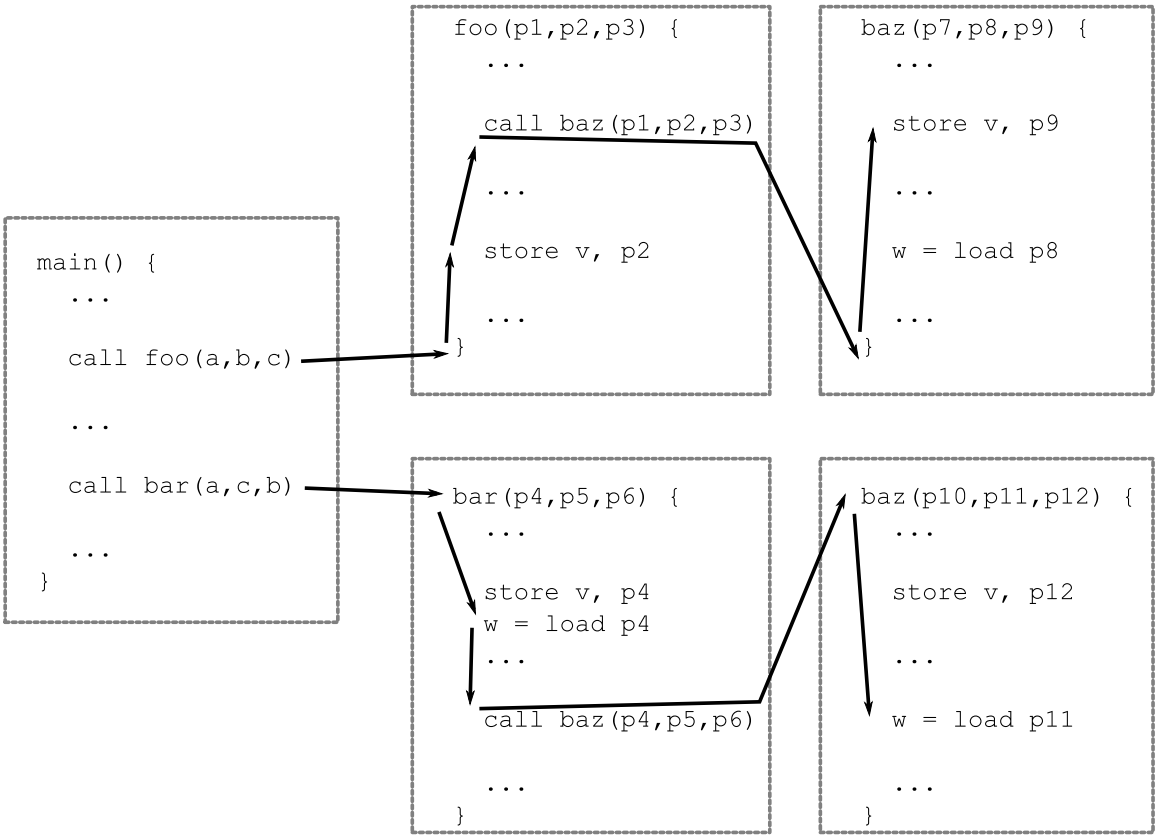


Figure A.1: Callsite depth-combinator searches for memory operations. Consider a query from `call foo` to `call bar`. Class `ReverseStoreSearch` visits downward-exposed `stores` in reverse order: `store v,p2` (with p2 = b) in `foo` and `store v,p9` (with p9 = p3 = c) in `baz`. Class `ForwardLoadSearch` visits upward-exposed `loads` in forward order: `w = load p11` (with p11 = p5 = b) in `baz`. It excludes `w = load p4` in `bar`. because `store v,p4` kills it. `call foo` flows to `call bar` only if either of `foo`'s downward-exposed `stores` flow to `bar`'s upward-exposed `load`.

instructions in callsites, starting from the roots of the dominator tree and visiting "earlier" instructions before the instructions they dominate. This iteration order is chosen to improve the performance of class `KillFlow`'s internal killing cache.

A.11 Global Malloc

- **Implementation:** class `GlobalMallocAA`
- **Tactic:** disprove aliasing; reachability.

- **Initialization costs:** linear in the number of instructions that *use* a global variable.
- **Cost / Query:** $O(A \log A)$, where A is the number of allocation sites captured into global variables.
- **Foreign Premise Queries / Query:** none.

During initialization, the "global malloc" algorithm identifies the subset of global variables whose addresses are never captured (see theme *simpler data flow on non-captured storage* in Section A.4). It also collects a set of values `stored` into non-captured global variables. It further classifies non-captured globals according to their definition sets: (a) all definitions come from an allocator routine (i.e. `malloc`), (b) no definitions come from an allocator, or (c) all others.

Global malloc answers queries pertaining to pointers loaded from global values. It reports no-alias between pointers when those pointers are loaded from a global in class (a) and a global in class (b) (because global classes (a) and (b) are disjoint). Also, it reports no-alias between pointers when both pointers are loaded from globals in class (a), and when the respective definition sets are disjoint.

A.12 Non-captured global

- **Implementation:** class `NoCaptureGlobalAA`
- **Tactic:** disprove aliasing; reachability.
- **Initialization costs:** none.
- **Cost / Query:** linear in the number of uses of a global variable.
- **Foreign Premise Queries / Query:** none.

The "non-captured global" algorithm reports no-alias between pointers when one pointer is loaded from memory and the other references a non-captured global variable. It reasons that non-captured globals are, by construction, those global variables whose address cannot be found in memory, and hence are disjoint from the loaded value.

A.13 Non-Captured Source

- **Implementation:** class `NoCaptureSrcAA`
- **Tactic:** disprove aliasing; reachability.
- **Initialization costs:** none.
- **Cost / Query:** linear in the number of uses of a global variable or allocation site.
- **Foreign Premise Queries / Query:** none.

The "non-captured source" algorithm identifies global variables or allocators whose address is never captured (see theme *simpler data flow on non-captured storage* in Section A.4). The algorithm exhaustively enumerates the set S of all uses of such objects by tracing intra-procedural register data flow.

The non-captured source algorithm reports no-alias among pointers when one pointer is in S and the other is not. When both pointers are in S, it reports no-alias if they reference different non-captured sources.

A.14 Unique Access Paths

- **Implementation:** class `UniquePathsAA`
- **Tactic:** disprove aliasing; reachability.
- **Initialization costs:** linear in size of program.

- **Cost / Query:** linear in size of points-to sets
- **Foreign Premise Queries / Query:** linear in size of points-to sets.

Unique Access Paths (UAP) is the example from Section 3.1.2. It reasons about a simple case of reachability, searching for global, heap, or stack objects whose address is never captured (see theme *simpler data flow on non-captured storage* in Section A.4). It collects points-to sets of values which are stored to those objects. The algorithm converts queries on pointers loaded from such paths into premise queries among the values in the points-to sets.

To improve performance, the sets of definitions are stored as move-to-front queues; this prioritizes premise which are not disproved, producing fewer premises for subsequent queries.

A.15 Array of Structures

- **Implementation:** class `ArrayOfStructures`
- **Tactic:** disprove aliasing; foreign premise queries.
- **Initialization costs:** none
- **Cost / Query:** effectively $O(1)$, scales linearly with the complexity of types.
- **Foreign Premise Queries / Query:** 1.

"Array of Structures" (AoS) is the example from Section 3.1.1. Given indexing operations `&a[`i_1`]...[`i_n`]` and `&b[`j_1`]...[`j_n`]`, it tries to prove $i_k \neq j_k$ with arithmetic and induction variable reasoning, tops a query to prove $a = b$, and if successful reports no dependence.

A.16 Scalar Evolution

- **Implementation:** class `SCEVAA`
- **Tactic:** disprove aliasing; induction variables.
- **Initialization costs:** none.
- **Cost / Query:** linear in size of function.
- **Foreign Premise Queries / Query:** none.

Class SCEVAA uses LLVM's scalar evolution analysis to find a symbolic representation of pointers as a function of loop induction variables. It subtracts these expressions and reports no-alias when the symbolic difference can be proved greater than the size of the memory access. Due to the design of LLVM's scalar evolution analysis, this algorithm is precise when both pointers are *affine* functions of induction variables.

A.17 SMTAA

- **Implementation:** class `ModuleSMTAA`
- **Tactic:** disprove aliasing; induction variables.
- **Initialization costs:** none.
- **Cost / Query:** linear in size of function **plus** an invocation of CVC3.
- **Foreign Premise Queries / Query:** none.

Class `SMTAA` reduces dependence analysis queries into Satisfiability-Modulo-Theories (SMT) sentences. It delegates them to the CVC3 solver [8]. The reduction is straightforward; in most cases it employs only the theory of Linear Integer Arithmetic (LIA), though

complicated queries will include statements from the theory of Non-Linear Integer Arithmetic (NIA) and the theory of Uninterpreted Functions (UF).

The reduction represents LLVM pointers and integer values as CVC3's unbounded `INT` values rather than `BITVECTOR(`n`)` values. This allows CVC3 to use the LIA theory and run faster in the common case. Unlike fixed-width LLVM integers, CVC3 `INT`s cannot overflow. This difference is unimportant, however, since integer overflow is undefined in C and the compiler is free to choose a behavior in response to undefined behavior.

The reduction models the allocation of global variables by choosing an arbitrary ordering of globals, and `ASSERT`ing that their range of memory addresses do not overlap. The reduction encodes some constraints drawn from the path-condition, for instance, that earlier control flow has established that iteration bounds are non-zero *before* reaching the loop header.

Finally, the reduction includes a query which asks the solver to refute the claim that the pointers are disjoint.

A.18 Sane Typing

- **Implementation:** classes `TypeSanityAnalysis` and `TypeAA`
- **Tactic:** disprove aliasing; type sanity.
- **Initialization costs:** linear in size of module.
- **Cost / Query:** constant.
- **Foreign Premise Queries / Query:** 0 or 1.

The "sane typing" algorithm identifies cases where the input program obeys a strong typing discipline, even though the C language or LLVM IR does not enforce such a discipline. In particular, it identifies a set S of types such that for all $\tau \in S$, if a pointer

expression p has declared type $\tau*$, then the value of p is either `null` or the address of an object of declared type τ. It calls such types *sane*. Several analysis algorithms build on sane typing analysis: "non-captured fields" (Section A.19), "acyclic" (Section A.20), "disjoint fields" (Section A.21), and "field malloc" (Section A.22).

`TypeSanityAnalysis` scans the module searching for reinterpretation casts (also called *type-punning*). The simplest examples of reinterpretation casts are LLVM's `bitcast`, `ptrtoint`, and `inttoptr` instructions. These instructions introduce differently-type-annotated names for their operands. This initialization scan also visits any construct which may conceal a reinterpretation cast. Code outside of the current compilation unit may reinterpret cast the values stored in non-`static` global variables, hence non-`static` globals are treated as a reinterpretation cast. Similarly, an externally-defined function may contain reinterpret casts, so externally-defined functions are treated as reinterpretation casts of their formal parameters and return values. When the scan discovers a reinterpretation to or from type τ, it marks type τ, and any types nested within τ, as *insane*.

After the scan, the set of *sane types* is represented implicitly as the complement of the set of insane types.

Dynamic heap allocation must be treated specially. Although LLVM global variables and stack allocations (LLVM's `alloca` instructions) include a type specification for the allocated storage, the LLVM IR does not include a first-order representation of heap allocation. Instead, calls to library routines such as `malloc` achieve heap allocation. In LLVM, these routines do not declare a type for the allocated objects but instead return pointers of type `i8*` (compare to C's use of `void*` as the type of generic pointers). Each heap allocation site is followed by a reinterpretation cast to the type as listed in the program's source code. Without special treatment, the types of all dynamically allocated objects would be marked insane.

`TypeSanityAnalysis` overcomes this problem by recognizing a common idiom generated by the compiler's front end. If the IR includes a call to an allocation routine, and

if the value returned by that allocation routine has a single use, and if that single use is a reinterpretation cast to type $\tau*$, then the entire sequence is treated as the allocation of an object with declared type τ instead of marking τ as insane.

Class `TypeAA` interprets sane types to service queries in two situations. The first simply codifies the observation that objects of different types are different. Although not true in general, it holds for sane types. Let p, q be pointers with declared types $\tau*$ and $\upsilon*$, respectively, where both τ and υ are sane types. p may-alias q only if $\tau = \upsilon$, or aggregate type τ contains (transitively) a field of type υ, or aggregate type υ contains (transitively) a field of type τ.

The second situation codifies the observation that objects of sane types do not partially overlap themselves; either they are disjoint or they have the *same* base address. Let p, q be pointers with declared types $\tau*$ and $\upsilon*$, respectively, where both τ and υ are sane types. Consider indexing expressions $P = \&p[i_1][i_2]\ldots[i_n]$ and $Q = \&q[j_1][j_2]\ldots[j_n]$. P cannot alias Q if $\tau = \upsilon$ and indices $i_k = j_k$ at all levels k of indexing and p does not alias q. `TypeAA` compares index expressions using simple arithmetic reasoning and uses a foreign premise query to determine whether p and q alias. Note that this rule is similar yet complementary to the reasoning used by the array of structures algorithm (Sections 3.1.1 and A.15).

A.19 Non-Captured Fields

- **Implementation:** classes `NonCapturedFieldsAnalysis` and `NoEscapeFieldsAA`
- **Tactic:** disprove aliasing; type sanity; reachability.
- **Initialization costs:** linear in size of program.
- **Cost / Query:** constant.

- **Foreign Premise Queries / Query:** 0 or 1.

The "non-captured fields" algorithm extends type sanity analysis (Section A.18) by identifying fields of sane types whose address is only used as an address for a `load` or `store` instruction. Note that this condition is slightly stronger than requiring the address is not captured. The algorithm uses a non-captured storage argument to reason about values loaded from such fields (see theme *simpler data flow on non-captured storage* in Section A.4). To be clear, it treats these fields in an object-insensitive manner, i.e. abstracting the given field within *all* objects of that type as a single storage location.

Class `NonCapturedFieldsAnalysis` scans the module to visit address computations of the form $f = \&b[k]$, where base pointer b has declared type $\tau*$. If τ is a sane type, such address computations derive the address f of the k-th field of some object of type τ. `NonCapturedFieldsAnalysis` considers all uses of f, either marking field $\tau :: k$ as captured or accumulating a points-to set $P(\tau :: k)$ of values stored into field $\tau :: k$ via the field pointer f. If the index k cannot be evaluated statically to a fixed field number, then all fields of τ are conservatively marked as captured.

Unlike `TypeSanityAnalysis`, note that `NonCapturedFieldsAnalysis` does not need to account for field-uses hidden in externally defined functions; the constraint that τ is a sane type precludes use by externally defined functions.

Class `NoEscapeFieldsAA` interprets the results of `NonCapturedFieldsAnalysis` to disprove aliasing. Consider two pointers constructed as $p_1 = \&b_1[f_1]$ and $p_2 = \&b_2[f_2]$, where base pointers b_1 and b_2 have declared types τ_1 and τ_2, respectively. Suppose that types τ_1, τ_2 are sane types, and that neither field $\tau_1 :: f_1$ nor field $\tau_2 :: f_2$ is captured. `NoEscapeFieldsAA` disproves aliasing between p_1 and p_2 in three cases:

1. $\tau_1 \neq \tau_2$: neither can be a subfield of the other because non-captured fields uses a *stronger* non-capture criterion (above);

2. $\tau_1 = \tau_2$ and $f_1 \neq f_2$: different fields are disjoint; or,

3. $\tau_1 = \tau_2$ and $f_1 = f_2$: the same fields alias only if their base objects alias. Issue a foreign premise query comparing b_1 to b_2.

A.20 Acyclic

- **Implementation:** class `AcyclicAA`
- **Tactic:** disprove aliasing; type sanity; reachability; shape analysis.
- **Initialization costs:** linear in size of input program.
- **Cost / Query:** linear in size of a function.
- **Foreign Premise Queries / Query:** none.

The "acyclic" algorithm extends type sanity (Section A.18) and non-captured fields (Section A.19). It identifies acyclic data structures as cases where recursive, non-captured fields are updated in a restricted manner. Specifically, it proves data structures acyclic via an inductive argument: *(base case)* newly allocated nodes are acyclic data structures, and *(inductive case)* assuming that a data structure is acyclic, attaching a new node at either end produces an acyclic data structure. Once a data structure is proved acyclic, it asserts that adjacent nodes in the data structure are disjoint.

A sane, aggregate type τ is *recursive* if it includes a non-captured field with declared type $\tau*$. Said fields are called *recursive fields*. For instance, the `next` and `prev` pointers in a textbook linked-list node are recursive fields; the `left_child` and `right_child` pointers of a textbook binary tree node are also recursive fields.

Class `AcyclicAA` performs an initialization analysis to identify recursive fields and then proves that certain recursive fields induce acyclic data structures. The proof verifies the inductive case for all mutations of recursive fields.

Initialization visits every instruction sequence that mutates recursive fields. In C, these sequences look like `b.k = v`. In LLVM IR, these sequences include a pointer expression

to compute a field's address $f = \&b[k]$ and an update `store` v, f. These sequences are further constrained such that: base pointer b has declared type $\tau*$; τ is a sane type; field $\tau :: k$ is a non-captured field; and field pointer f has declared type $\tau **$. Observe that type sanity and non-captured fields guarantee that the compiler identifies all such sequences. Hence, this scan is equivalent to universal quantification over all mutations.

Next, class `AcyclicAA` verifies the inductive hypothesis on each mutation. Considering the mutation `b.k = v`, there are three cases which cannot introduce a cycle.

1. `b` is a newly allocated object: this mutation pushes a new node at the beginning of a linked-list or at the root of a tree.

2. `v` is the unique use of a newly-allocated object: this mutation pushes a new node at the end of a linked-list or at the leaf of a tree.

3. `v` is `null`: this mutation cuts a linked list or tree.

Any other mutation potentially introduces a cycle. Since class `AcyclicAA` cannot validate the inductive proof for such case, it conservatively marks the field $\tau :: k$ as cyclic.

The acyclic algorithm uses this information to disprove aliasing among pointers to recursive types when one pointer is derived from the other by traversing acyclic fields. Traversals take a few forms:

- **Immediate traversal:** Suppose pointer p is declared with sane type $\tau*$ and is computed from $p =$ `load` f. Further, suppose that $f = \&q[k]$ where base pointer q is declared with sane type $\tau*$ and the field $\tau :: k$ is an acyclic field. Then pointer p is an *immediately traversal* of q.

- **Intra-iteration traversal:** *Intra-iteration traversal* is the transitive closure of the immediate traversal relation. If p is an immediate traversal of q, then p is an intra-iteration traversal of q. Further, if p is an intra-iteration traversal of q, and if q is an immediate traversal of r, then p is an intra-iteration traversal of r.

- **Inductive traversal:** Suppose that a loop L includes a Φ-node ϕ declared with sane type $\tau*$. Further, suppose that ϕ's incoming values along all backedges of L are intra-iteration traversals of ϕ. Then ϕ is an *inductive traversal* in L.

- **Loop-carried traversal:** Suppose that ϕ is an inductive traversal in L, and that p is an intra-iteration traversal of ϕ. Then p is a *loop-carried traversal* of ϕ in L.

When class `AcyclicAA` receives an intra-iteration may-alias query, it reports no-alias if it demonstrates that one pointer is an intra-iteration traversal of the other. When `AcyclicAA` receives a loop-carried may-alias query with respect to loop L, it reports no-alias if it demonstrates that one pointer is a loop-carried traversal of the other in L.

A.21 Disjoint Fields

- **Implementation:** class `DisjointFieldsAA`

- **Tactic:** disprove aliasing; reachability; shape analysis.

- **Initialization costs:** linear in size of program.

- **Cost / Query:** linear in size of points-to sets.

- **Foreign Premise Queries / Query:** none.

The "disjoint fields" algorithm extends type sanity (Section A.18) and non-captured fields (Section A.19). By collecting a points-to set of values stored into non-captured fields, class `DisjointFieldsAA` asserts that pointers loaded from non-captured fields are disjoint if their respective points-to sets are disjoint.

A.22 Field Malloc

- **Implementation:** class `FieldMallocAA`

- **Tactic:** disprove aliasing; reachability; shape analysis; foreign premise queries.
- **Initialization costs:** linear in size of program.
- **Cost / Query:** constant.
- **Foreign Premise Queries / Query:** 1

The "field malloc" algorithm extends type sanity (Section A.18) and non-captured fields (Section A.19). It then considers every store into a non-captured field. If such stores are the unique capture of another allocation, then said allocation can only be accessed by loading from the non-captured fields. We call such a field the *unique name* for an allocation.

Class `FieldMallocAA` simplifies queries where both pointer operands are loads from a unique name field. It reasons that the loaded pointers alias only if they are loaded from the same object. Thus, it reports no alias if a foreign premise query establishes that the base objects do not alias.

Bibliography

[1] ISO/IEC 9899-201x Programming Languages—C, Committee Draft, 2011.

[2] The OpenMP API specification. http://www.openmp.org.

[3] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the American Federation of Information Processing Societies (AFIPS) April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[4] L. O. Andersen. Program analysis and specialization for the C programming language, May 1994.

[5] Andrew W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[6] Utpal Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Norwell, MA, 1993.

[7] Utpal Banerjee. *Loop Parallelization*. Kluwer Academic Publishers, Boston, MA, 1994.

[8] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[9] Anasua Bhowmik and Manoj Franklin. A fast approximate interprocedural analysis for speculative multithreading compilers. In *Proceedings of the 17th annual international conference on Supercomputing*, ICS '03, pages 32–41, New York, NY, USA, 2003. ACM.

[10] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Thresher: precise refutations for heap reachability. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI '13, pages 275–286, New York, NY, USA, 2013. ACM.

[11] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 243–262, New York, NY, USA, 2009. ACM.

[12] Matthew J. Bridges. *The VELOCITY Compiler: Extracting Efficient Multicore Execution from Legacy Sequential Codes*. PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States, November 2008.

[13] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, 2008.

[14] Peng-Sheng Chen, Yuan-Shin Hwang, Roy Dz-Ching Ju, and Jenq Kuen Lee. Interprocedural probabilistic pointer analysis. volume 15, pages 893–907, Piscataway, NJ, USA, 2004. IEEE Press.

[15] T. Chen, J. Lin, W.C. Hsu, and P.C. Yew. An empirical study on the granularity of pointer analysis in C programs. *Languages and Compilers for Parallel Computing (LCPC)*, pages 157–171, 2005.

[16] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17, 1995.

[17] R. Cytron. DOACROSS: Beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing (ICPP)*, pages 836–884, 1986.

[18] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[19] Jeff Da Silva and J. Gregory Steffan. A probabilistic pointer analysis for speculative optimizations. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 416–425, New York, NY, USA, 2006. ACM Press.

[20] Francis H. Dang, Hao Yu, and Lawrence Rauchwerger. The R-LRPD test: Speculative parallelization of partially parallel loops. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, pages 20–29, 2002.

[21] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, July 1987.

[22] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Proceedings of the 7th International Symposium on Static Analysis (SAS)*, pages 175–198, London, UK, UK, 2000. Springer-Verlag.

[23] Free Software Foundation. *man gcc(1)*, 2008.

[24] Freddy Gabbay and Avi Mendelson. Can program profiling support value prediction? In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, pages 270–280, Washington, DC, USA, 1997. IEEE Computer Society.

[25] GCC Development Mission Statement (1999-04-22), April 1999. http://gcc.gnu.org/gccmission.html.

[26] R. Ghiya and L. J. Hendren. Is it a Tree, DAG, or Cyclic Graph? In *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 1996.

[27] Rakesh Ghiya, Daniel Lavery, and David Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)*, pages 47–58. ACM Press, 2001.

[28] Bolei Guo, Neil Vachharajani, and David I. August. Shape analysis with inductive recursion synthesis. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 256–265, June 2007.

[29] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation (PLDI)*, pages 24–34, New York, NY, 2001.

[30] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, Fourth Edition*. Morgan Kaufmann, 2006.

[31] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2001.

[32] Michael Hind and Anthony Pioli. Evaluating the effectiveness of pointer alias analyses. In *Science of Computer Programming*, pages 31–55, 1999.

[33] S. Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems*, 19(1), January 1997.

[34] S. Horwitz, J. Prins, and T. Reps. On the adequacy of program dependence graphs for representing programs. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 146–157, 1988.

[35] Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In *In proceedings of the Fourtheenth International Conference on Software Engineering (CSE)*, pages 392–411, 1992.

[36] Jialu Huang, Thomas B. Jablin, Stephen R. Beard, Nick P. Johnson, and David I. August. Automatically exploiting cross-invocation parallelism using runtime information. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, Washington, DC, USA, 2013. IEEE Computer Society.

[37] Thomas B. Jablin, James A. Jablin, Prakash Prabhu, Feng Liu, and David I. August. Dynamically Managed Data for CPU-GPU Architectures. In *Proceedings of the 2012 International Symposium on Code Generation and Optimization*, April 2012.

[38] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. Automatic CPU-GPU communication management and optimization. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[39] Yunlian Jiang and Xipeng Shen. Adaptive speculation in behavior-oriented parallelization. In *Parallel and Distributed Processing, 2008. IPDPS 2008.*, 2008.

[40] Nick P. Johnson, Hanjun Kim, Prakash Prabhu, Ayal Zaks, and David I. August. Speculative separation for privatization and reductions. *Programming Language Design and Implementation (PLDI)*, June 2012.

[41] Nick P. Johnson, Taewook Oh, Ayal Zaks, and David I. August. Fast condensation of the program dependence graph. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI '13, pages 39–50, New York, NY, USA, 2013. ACM.

[42] Maurice G. Kendall. *Rank Correlation Methods*. Charles Griffin and Company, Limited, London, 1948.

[43] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[44] Hanjun Kim. *ASAP: Automatic Speculative Acyclic Parallelization for Clusters*. PhD thesis, Princeton, NJ, USA, 2013.

[45] Hanjun Kim, Nick P. Johnson, Jae W. Lee, Scott A. Mahlke, and David I. August. Automatic speculative doall for clusters. *International Symposium on Code Generation and Optimization (CGO)*, March 2012.

[46] Hanjun Kim, Arun Raman, Feng Liu, Jae W. Lee, and David I. August. Scalable speculative parallelization on commodity clusters. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010.

[47] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. SD^3: A scalable approach to dynamic data-dependence profiling. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '10, pages 535–546, Washington, DC, USA, 2010. IEEE Computer Society.

[48] D. J. Kuck, Y. Muraoka, and S. C. Chen. On the number of operations simultaneously executable in fortran-like programs and their resulting speedup. *IEEE Transactions on Computers*, C-21:1293–1309, December 1972.

[49] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, 1992.

[50] Samuel Larsen, Emmett Witchel, and Saman Amarasinghe. Increasing and detecting memory address congruence. In *In International Conference on Parallel Architectures and Compilation Techniques*, pages 18–29, 2002.

[51] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the Annual International Symposium on Code Generation and Optimization (CGO)*, pages 75–86, 2004.

[52] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, California, June 2007.

[53] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 270–282, New York, NY, USA, 2002. ACM.

[54] Xavier Leroy and Sandrine Blazy. Formal verification of a c-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41:1–31, 2008.

[55] Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1):3:1–3:53, October 2008.

[56] Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai, and Sun Chan. A compiler framework for speculative analysis and optimizations. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 289–299, New York, NY, USA, 2003. ACM.

[57] Feng Liu, Soumyadeep Ghosh, Nick P. Johnson, and David I. August. Generating high-performance accelerators via coarse-grained pipeline parallelism. In *Design and Automation Conference*, DAC '14. ACM, 2014.

[58] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: a TLS compiler that exploits program structure. In *PPoPP '06: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 158–167, 2006.

[59] Thomas R. Mason. Lampview: A loop-aware toolset for facilitating parallelization. Master's thesis, Department of Electrical Engineering, Princeton University, Princeton, New Jersey, United States, August 2009.

[60] Robert Muth and Saumya Debray. On the complexity of flow-sensitive dataflow analyses. In *In Proc. ACM Symp. on Principles of Programming Languages*, pages 67–80. ACM Press, 2000.

[61] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1:245–257, 1979.

[62] Guilherme Ottoni. *Global Instruction Scheduling for Multi-Threaded Architectures*. PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States, 2008.

[63] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th Annual*

IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 105–118, 2005.

[64] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, San Francisco, CA, 2nd edition, 1998.

[65] Prakash Prabhu, Soumyadeep Ghosh, Yun Zhang, Nick P. Johnson, and David I. August. Commutative set: A language extension for implicit parallel programming. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[66] Prakash Prabhu, Thomas B. Jablin, Arun Raman, Yun Zhang, Jialu Huang, Hanjun Kim, Nick P. Johnson, Feng Liu, Soumyadeep Ghosh, Stephen Beard, Taewook Oh, Matthew Zoufaly, David Walker, and David I. August. A survey of the practice of computational science. *Proceedings of the 24th ACM/IEEE Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, November 2011.

[67] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of Supercomputing 1991*, pages 4–13, November 1991.

[68] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the Fifteenth International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[69] Arun Raman, Ayal Zaks, Jae W. Lee, and David I. August. Parcae: A system for flexible parallel execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, New York, NY, USA, 2012. ACM.

[70] Easwaran Raman. *Parallelization Techniques with Improved Dependence Handling*. PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States, June 2009.

[71] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew Bridges, and David I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the Annual International Symposium on Code Generation and Optimization (CGO)*, 2008.

[72] Easwaran Raman, Neil Vachharajani, Ram Rangan, and David I. August. Spice: speculative parallel iteration chunk execution. In *CGO '08: Proceedings of the 2008 International Symposium on Code Generation and Optimization*, pages 175–184, New York, NY, USA, 2008. ACM.

[73] Norman Ramsey, João Dias, and Simon Peyton Jones. Hoopl: A modular, reusable library for dataflow analysis and transformation. In *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, pages 121–134, New York, NY, USA, 2010. ACM.

[74] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. Decoupled software pipelining with the synchronization array. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 177–188, September 2004.

[75] Lawrence Rauchwerger and David Padua. The Privatizing DOALL test: A run-time technique for DOALL loop identification and array privatization. In *Proceedings of the 8th international conference on Supercomputing*, ICS '94, pages 33–43, New York, NY, USA, 1994. ACM.

[76] Lawrence Rauchwerger and David A. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel Distributed Systems*, 10:160–180, February 1999.

[77] M. Sagiv, T. Reps, and R.Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–31, January 1996.

[78] Standard Performance Evaluation Corporation. http://www.spec.org.

[79] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 59–76, New York, NY, 2005.

[80] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[81] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving value communication for thread-level speculation. In *Proceedings of the 8th International Symposium on High Performance Computer Architecture*, pages 65–80, February 2002.

[82] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[83] A. J. Thadhani. Factors affecting programmer productivity during application development. *IBM Systems Journal*, 23(1):19 –35, 1984.

[84] Chen Tian, Min Feng, Vijay Nagarajan, and Rajiv Gupta. Copy or discard execution model for speculative parallelization on multicores. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 330–341, Washington, DC, 2008. IEEE Computer Society.

[85] Peng Tu and David A. Padua. Automatic array privatization. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 500–521, 1994.

[86] Neil Vachharajani. *Intelligent Speculation for Pipelined Multithreading*. PhD thesis, Department of Computer Science, Princeton University, Princeton, New Jersey, United States, November 2008.

[87] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. Speculative decoupled software pipelining. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 49–59, Washington, DC, USA, 2007. IEEE Computer Society.

[88] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. The Paralax infrastructure: Automatic parallelization with a helping hand. In *Proceedings of the 19th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 389–400, 2010.

[89] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *SIGOPS Oper. Syst. Rev.*, 27:203–216, December 1993.

[90] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, (ICSE), pages 439–449, Piscataway, NJ, 1981.

[91] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, (PLDI), pages 131–144, New York, NY, 2004.

[92] *The Wisconsin Program-Slicing Tool, Version 1.1*, 2000. http://research.cs.wisc.edu/wpis/slicing_tool/.

[93] Hongtao Yu, Hou-Jen Ko, and Zhiyuan Li. General data structure expansion for multi-threading. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 243–252, New York, NY, USA, 2013. ACM.

[94] Jianzhou Zhao. *Formalizing the SSA-based compiler for verified advanced program transformations*. PhD thesis, Department of Computer Science, University of Pennsylvania, Philadelphia, PA, United States, Jan 2013.

[95] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *In 39th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2012.

[96] Xin Zheng and Radu Rugina. Demand-driven alias analysis for C. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 197–208, New York, NY, 2008.