Finding and Exploiting Parallelism with Data-Structure-Aware Static and Dynamic Analysis

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Christopher Ian Fallin

B.S. Computer Engineering, May 2009, University of Notre Dame
M.S. Electrical and Computer Engineering, Dec 2011, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

May 2019

Acknowledgments

Having been at CMU nearly a decade, worked on multiple projects in multiple research groups, and seen many friends and fellow researchers come and go, I have many more acknowledgments to give than these few pages will allow. Nevertheless, as I have learned, when one faces an impossible problem, sometimes with enough effort a reasonable solution can appear. So here's an attempt.

First, my advisors, Profs. Todd Mowry and Phil Gibbons, have been extraordinarily helpful and supportive in various and complimentary ways. Through their experience, they helped to chart a path for this research, focus on the most interesting questions, and refine the concepts and abstractions we invented into what I present here. They both granted significant hands-off freedom to me as I tried various approaches to the autoparallelization problem and struggled through some false starts before discovering the foundations of what eventually worked. At the same time, our weekly meetings consistently kept me on my toes with incisive questions and genuinely helpful technical insights. I thank Todd in particular for being willing to take on a "slightly used" graduate student when I first considered returning to CMU in 2014 and finally arrived in 2015, and for setting the direction and tone of the "semantic lifting" project. And I thank Phil for being willing to join our weeklies in my first year and consistently surprising me with the detail-orientedness of his questions and insights, suggestions based on his systems and algorithms experience, and for connecting us more closely with the PDL community. Both have influenced my taste for interesting problems and have provided an encouraging, trouble-free environment in which to solve them; for that, I am very thankful.

My thesis committee members Prof. Jonathan Aldrich, Prof. Kayvon Fatahalian and Prof. Luis Ceze all contributed useful feedback and interesting discussions, insights and alternative perspectives on my work, and at each stage were eager to help. I thank them all for taking the time to review this thesis and providing their very valuable input.

My fellow research-group members were essential to my success, and good friends as well. Pratik Fegade arrived on this project in 2016 and very quickly began making valuable technical contributions; since then he has been right in the trenches with me, providing a sounding board, rapidly grasping new ideas, turning over gnarly technical issues and inventing solutions, and pushing the analysis and evaluation infrastructure forward, one paper-submission sprint at a time. He has an impressive ability to zero in on edge-case correctness bugs; if you want a correct analysis implementation, first build a slightly wrong version, and then wait for Pratik's inevitable pull request. Dominic Chen joined weekly meetings with the two of us and provided excellent feedback as well. I appreciate very much the support that our group has provided over the last several years.

I thank the Parallel Data Lab (PDL) community, both faculty/students and industry members, who welcomed our group at the annual PDL retreats in beautiful Bedford Springs and provided interesting and insightful perspective on our work from a systems point of view. Thanks also to the weekly CIC reading group, which has been a fruitful source of interesting discussion and research stimulus.

The work in this thesis was built on the shoulders of some particularly capable giants: the Doop Java points-to analysis, the Soot Java compiler framework, and the Soufflé Datalog compiler. Thanks to the respective research teams for building these useful tools, and in particular to George Kastrinis and Prof. Yanniss Smaragdakis for answering a few questions about Doop.

Other students served to keep me sane over the ups and downs of the past four years. I am particularly grateful to the denizens of GHC 6207, who became good friends. Ryan Kavanagh, Costin Bădescu, and Anson Kahng happily engaged in mutual time wastage, Wikipedia rabbit-hole traversal, and various other schenanigans that were nevertheless the most enjoyable parts of my time in the office. Thanks for the camaraderie and the free-food-table alerts too. Thanks also to former inhabitants Thomas Kim, Vaishnavh Nagarajan, Kartik Gupta, and recent arrival Brian Schwedock, for tolerating and/or tuning out our banter. (Thanks also to the CS Department for graciously allowing me to occupy one of the most strategically-located CS grad student offices in Gates, despite my status as an ECE student.)

My past life in "PhD version one," as well as my full-time stint in industry, also heavily influenced my interests and orientation toward research. I acknowledge my former advisor, Prof. Onur Mutlu, for originally bringing me to CMU and providing opportunities for me to gain experience and explore up and down the systems stack. He influenced my taste for interesting and fundamental research questions, and taught me how to distill often complex and subtle ideas down to key insights with clear descriptions. These skills have proved immensely useful throughout the years. Prof. James Hoe provided valuable advice at several key points, and helped me work out my return to CMU as well.

Chris Wilkerson at Intel was a willing mentor who was generous with his time and encouragement from my first Intel internship onward, and especially in my core microarchitecture work. Brian Hirano of Oracle was always happy to provide interesting insights and observations in weekly phone calls. Eugene Gorbatov enthusiastically mentored my work during my first internship. During my time on the Intel Oregon core microarchitecture team, Mark Dechene brought me up to speed as an intern, and Rob Chappell was the most fair-minded, decent, and helpful manager one could imagine during my full-time work. Jared Stark taught me all I know about branch predictors and then some, and gave me a new level of appreciation for the power of clever engineering tricks. At Google, Jayant Kolhe was an unfailingly helpful manager who took my desire to return to CMU in stride and made my transition as easy as possible. Josh Haberman was happy to engage in researchy discussions, keeping life interesting among a sea of low-level implementation work, and Rohan Anil's relentless optimization grind reinforced the lesson that one sometimes finds good numbers by simply doing the work in a thousand needed places. All of these industry experiences prepared me in some way or another to build more than I thought possible in this thesis work.

Fellow SAFARI group members during my first CMU epoch became lifelong friends, and I owe much of my "growing up" to this odd research family of sorts.

Yoongu Kim was the pioneer and wise older brother, the first student of the group and a year above my cohort, whose constant good humor and easy-going nature kept us all sane. Chris Craik was my early research partner and good friend in our first year, before he left for the (metaphorically) greener pastures of California. Vivek Seshadri, the squash player with a grad-student habit, was a brilliant bubbling fountain of ideas, each of them a potential thesis unto itself, who inspired us all. Lavanya Subramanian was an excellent fellow TA and quals cohort-mate whose diligence and willingness to help were inspiring. Justin Meza's research brilliance and superb presentation aesthetics and writing skills were surpassed only by his friendly welcome to all in our group (and his willingness to help me consume my homebrew). Rachata Ausavarungnirun was an extraordinarily talented cook and good friend who fed us on many occasions, and his research persistence and willingness to assist any project were invaluable. Kevin Chang worked well with Rachata to take over some of my earlier interconnects work, and was an excellent collaborator and friend. Jamie Liu is probably the smartest person I've met, and his ability to produce two ISCA papers in three semesters and then escape to the good life is still mind-boggling. Ben Jaiyen was likewise a friendly and brilliant addition to our group before he too found the good life out west. The fact that I accidentally became Jamie and Ben's literal next-door neighbor in Mountain View was simply an added bonus.

Gennady Pekhimenko was a brilliant researcher, with time for a conversation with anyone despite having a hand in many projects. I thank Gena in particular for convincing me to consider returning to CMU. HanBin Yoon brought a lot of good humor to our group, and somehow survived his time with us despite several nearby electric fans. Donghyuk Lee knew more than any mortal about DRAM internals, and was always willing to make use of his experience to help everyone around him. Hongyi Xin did inspiring work in DNA sequencing, which taught me to always appreciate the real-world impact our research can have. I also thank him for the fascinating lessons on Chinese history and politics. Yixin Luo was a friendly and helpful addition to our group during my last year in SAFARI. Samira Khan was a skilled post-doc with much wisdom and experience to share who thankfully was not too cool to become "one of us." Greg Nazario and Xiangyao Yu were two wonderful undergraduate interns who helped significantly with my research in summer 2011. Ross Daly and Tyler Huberty were incredibly talented bachelors/masters students who contributed significantly to work in our group (I thank Tyler especially for sharing my passion for hikes in the Santa Cruz Mountains). Rachael Harding likewise was a very skilled undergraduate student who contributed to an early project with Yoongu and me, and added to our late-night camaraderie in the lab.

In the extended CMU/CALCM family, Michael Papamichael and Eric Chung of Prof. James Hoe's group went out of their way to welcome SAFARI as we grew into their space, and both of them provided guidance to me in my early years. George Nychis was an excellent mentor, collaborator and friend who took me under his wing in my first semester, leading to a few interesting publications and easing my transition to CMU significantly. Gabe Weisz was also welcoming and friendly, and was an excellent partner on our compilers course project. Michelle Goodstein and Tunji Ruwase, fellow students of Todd's, were both a friendly and encouraging presence when we moved to CIC. Brendan Meeder, fellow PhD student and housemate along with Chris Craik, was always willing to commiserate. Finally, outside of CMU but within the academic family, Rustam Miftakhutdinov, Eiman Ebrahimi, Khubaib, and Carlos Villavieja of Prof. Yale Patt's group at UT-Austin were always eager to talk research and provide encouragement to their academic relatives, both remotely and whenever we both found ourselves at Intel.

Many staff members in ECE, and several in CSD as well, worked hard to help keep my work running smoothly. I thank the three generations of ECE PhD program advisors I have witnessed: Elaine Lawrence, Samantha Goldstein, and Nathan Snizaski. Thanks too to Marilyn Patete and Jennifer Gabig formerly of ECE, and Deb Cavlovich, Diana Hyde, Angela Miller, Angela Luck, and Anthony Moreino in CSD.

During my undergraduate days, Profs. Peter Kogge and Jay Brockman (a fellow CMU-ECE PhD!) were happy to take me as far as I wanted to go in computer architecture and strongly encouraged my graduate-school ambitions. I am grateful for their help. Prof. Pat Flynn provided me with the opportunity to do undergrad research as well, which was very valuable experience.

I am particularly grateful for the influence of Dr. John Gorman of Jesuit High School in Portland, Oregon, my high-school math teacher for four years who is likely one of the few such instructors to eagerly go into number theory and combinatorics, group theory and abstract algebra, public-key cryptography, and myriad other topics, all with painstakingly prepared course notes in LaTeX, plus extra time after school and at the local Starbucks. He taught me how to think mathematically, develop proofs, pursue research ideas, and to appreciate what can be done with careful, hard thought. Thank you! Other friends were also influential early on; in particular, thanks go to Keshav Kini, who introduced me to Microsoft QBASIC in 1997 and condemned me irreedemably to this path.

I am eternally thankful for the environment and the encouragement my parents John and Nancy Fallin provided from as early as I can remember – from the soldering iron and first spool of wire (22 AWG, stranded), to the proximity to computing equipment of all sorts and free rein to install GNU/Linux, to tolerating my monopolization of the 56k dialup, and providing probably too much leeway whenever I preferred an interesting debugging problem over a prompt response to the dinner call, they provided a world where I was encouraged to *think* and *learn*, with all the opportunities I could have ever wanted. Thanks to my brother

Brian for tolerating an excessively nerdy brother. Finally, I am grateful to my girlfriend Ann Shue, whom I met soon after my return to CMU and whose constant encouragement has made all the difference as we both work to complete our training and start "real life."

The work in this thesis was funded by NSF, Intel, and the Parallel Data Lab (PDL) at CMU. My earlier time at CMU was funded by an NSF Graduate Research Fellowship, an SRC Fellowship, a Qualcomm Innovation Fellowship honorable mention, and the Bertucci Fellowship.

I thank Coffee Tree Roasters (Squirrel Hill and Shadyside), Commonplace Coffee, and Tazza d'Oro for the essential task of keeping my caffeine receptors caffeinated during this work.

Chris Fallin
February 7, 2019
(graduate school day 3468)

And so, his thesis completed,
His data and code he deleted.
'Til along came advisor
And said 'twould be wiser
If results could again be repeated!

Abstract

Maximizing performance on modern multicore hardware demands aggressive optimizations. Large amounts of legacy code are written for sequential hardware, and parallelization of this code is an important goal. Some programs are written for one parallel platform, but must be periodically updated for other platforms, or updated with the existing platform's changing characteristics – for example, by splitting work at a different granularity or tiling work to fit in a cache. A programmer tasked with this work will likely refactor the code in ways that diverge from its original implementation's step-by-step operation, but nevertheless computes correct results.

Unfortunately, because modern compilers are unaware of the higher-level structure of a program, they are largely unable to perform this work automatically. First, they generally preserve the operation of the original program at the language-semantics level, down to implementation details. Thus parallelization transforms are often hindered by false dependencies between data-structure operations that are semantically commutative. For example, reordering (e.g.) two tree insertions would produce a different program heap even though the tree elements are identical. Second, by analyzing the program at this low level, they are generally unable to derive invariants at a higher level, such as that no two pointers in a particular list alias each other. Both of these shortcomings hinder modern parallelization analyses from reaching the performance that a human programmer can achieve by refactoring.

In this thesis, we introduce an enhanced compiler and runtime system that can parallelize sequential loops in a program by reasoning about a program's *high-level semantics*. We make three contributions. First, we enhance the compiler to reason about *data structures* as *first-class values*: operations on maps and lists, and traversals over them, are encoded directly. *Semantic models* map library data types that implement these standard containers to the IR intrinsics. This enables the compiler to reason about commutativity of various operations, and provides a basis for deriving data-structure invariants. Second, we present *distinctness analysis*, a type of static alias analysis that is specially designed to derive results for parallelization that are just as precise as needed, while remaining simple and widely applicable. This analysis discovers non-aliasing across loop iterations, which is a form of flow-sensitivity that nevertheless is much simpler than past work's closed-form analysis of linear indexing of data structures. Finally, for cases when infrequent occurrences rule out a loop parallelization, or when static analysis cannot prove the parallelization to be safe, we leverage *dynamic checks*. Our hybrid static-dynamic approach extends the logic of our static alias analysis to find a set of checks to perform at runtime, and then uses the implications of these checks in its static reasoning. With the proper runtime techniques, these dynamic checks can be used to parallelize many loops without any speculation or rollback overhead.

This system, which we have proven sound, performs significantly better than prior standard loop-parallelization techniques. We argue that a compiler and runtime system with built-in data-structure primitives, simple but effective alias-analysis extensions, and some carefully-placed dynamic checks is a promising platform for macro-scale program transforms such as loop parallelization, and potentially many other lines of future work.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

There once was a domain-spec'fic language
Whose use often caused painful anguish:
Though structures, parallel,
Were done fairly well,
Its expressive powers had languished.

Modern software often does not make use of the available parallelism on ubiquitous multicore and other recently-introduced parallel hardware platforms. This is due to at least three reasons. First, refactoring existing code in imperative languages to take advantage of parallelism is difficult, because these languages often allow arbitrary side-effects and encourage low-level heap object manipulation. Second, adapting code to compute in parallel can be difficult if the available concurrency-control abstractions are too primitive. On many platforms, only OS-level threads are available, and there are few built-in language features for expressing data parallelism, for example. Finally, tuning parallel performance for a particular platform is difficult because the best performance may arise from very different settings, or design choices, on each platform.

1.1 The Problem: Levels of Program Understanding

A human programmer approaching the task of refactoring a program generally works to understand the program's algorithm or task at a *high level* first – for example, to sort a list, to multiply two arrays, or to process independent work-items with the results merged into a key-value map – and then to make various implementation decisions to accomplish that task. The programmer may choose data structures, control flow and parallelization strategies, etc., in order to enable certain goals, such as spreading work across cores. The high-level specification is separate from, and constant across, the large design space of possible implementations.

```
          HashMap map;
          ArrayList list;

(i)       for (int i = 0; i < 100; i++)
            list.add(i);

(ii)      for (Integer i : list)
            map.put(i, new Parent());

(iii)     for (Integer i : map.keySet())
            map.get(i).childPtr = new Child();

(iv)      for (Integer i : list)
            map.get(i).childPtr.field = i;
```

Figure 1.1: Example program: we consider whether the four loops are *parallelizable*.

When refactoring *existing code*, in particular, a human programmer first *reverse-engineers* the program to recover this high-level understanding. The program as written specifies many details that are not essential to the core algorithm. For example, it may contain data-structure implementations and intricate code to maintain invariants on those data structures. It may also contain traversals over that data or particular control-flow idioms such as a work-queue loop or structural recursion over a tree. It may use ad-hoc intermediate data structures. Programmers often have a library of idioms that they understand through experience, including all of the above. Additionally, when developing the code, the programmer establishes and adheres to high-level invariants. For example, the object pointers in a list may all refer to separate objects. Automatic analysis of such patterns might be even more difficult due to corner-case bugs: an invariant may not technically be true, though the programmer understands the intent anyway. The programmer then has the ability to synthesize these facts into a high-level summary. For example, a loop may traverse over a list of objects, each of them separate, and perform independent work for each object. This particular example is the common `map()` functional-programming idiom, and if recognized as such, its iterations can easily be parallelized.

1.2 The Problem Illustrated: Loop Parallelization

To illustrate the problem concretely, let us focus now on *loop parallelization*. By *loop parallelization*, we mean the problem of transforming a loop in a program so that its iterations operate completely in parallel: they may be spawned in an arbitrary order and may execute concurrently, on multiple cores, with a barrier at the loop exit that awaits completion of all iterations. We say that a loop is *parallelizable* if program state after the exit of the loop is identical[1] whether or not the loop has been parallelized.

Consider the small Java program in Fig. 1.1. The program consists of four loops: it (i) iterates over the

[1]Note that a definition of "identical" state is required here, and a stricter definition may restrict some parallelization. For example, identical may mean observationally identical as viewed through the data structure's query API. Or, more strongly, it may require a byte-for-byte identical memory image.

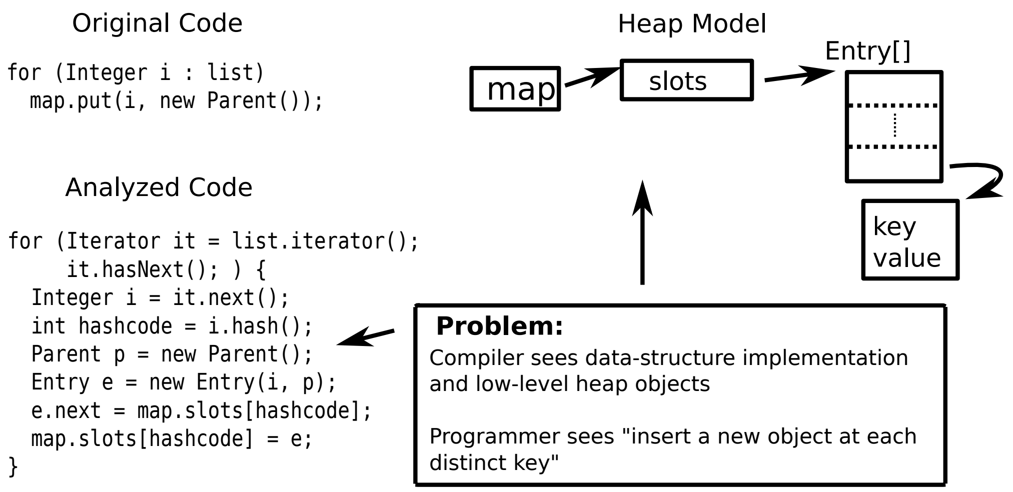
integers from 0 to 99, adding them to a list; (ii) iterates over this list, inserting key-value pairs into a map; (iii) iterates over the values in the map, setting a field; and (iv) iterates over the values again, setting a field of a child object.

In considering whether each of the four loops is parallelizable, we can quickly make several observations. First, all four loops mutate a *shared data structure*, so a simple definition of parallelizability that requires non-aliasing memory accesses will disqualify all four loops. However, if we consider *API-level equivalence*, which in this program means that the map and list have the same contents as viewed through the API, then every loop except the first is parallelizable, as long as we insert appropriate locking to ensure thread-safe accesses to the list and map. This can be seen by observing that the second, third, and fourth loops operate only on objects keyed by a different integer `i` each iteration. Finally, if only `map` is live at the end of the four loops, and not `list`, then the first loop is parallelizable as well: it does not matter in which order we process keys in the second and fourth loops, thus the program output is equivalent for any order of elements in `list`, and the append operations in the first loop can complete out of order (in parallel iterations).

Consider, now, how a static compiler analysis is likely to analyze this program. Fig. 1.2 illustrates this view for the second loop of Fig. 1.1, showing both the code as seen by the compiler and its likely model of the program heap. First, due to the separation of concerns between the compiler and the standard library, the compiler does not have any concept of a data structure such as `HashMap`. Rather, it analyzes the standard library like any other code. In particular, it sees the data structure as just another type of heap object, and it analyzes the data-structure methods as ordinary methods. (We have shown simplified versions of these details in the figure.)

This separation is generally a sound design decision, because it requires the compiler to consider fewer primitive operations, which simplifies its analysis and code-generation. However, in this case, it obscures the intent of the code in a way that hinders useful results. For example, in this case, it will almost certainly judge the loop to be non-parallelizable because it cannot prove that the hash-table array accesses in each iteration do not collide. Indeed, they may have a true conflict if two keys hash to the same hash-table slot. This is a true data dependency between loop iterations, and an analysis that deduces this dependency would inhibit parallelization because of it.

To conclude that the above loop is parallelizable, the compiler needs to understand two things. First, it must understand that map insertions are *commutative* as long as the keys for each insert are different. In other words, inserting $k_1 \Rightarrow v_1$ and $k_2 \Rightarrow v_2$ into `map` in either order should yield an equivalent state when $k_1 \neq k_2$. If the compiler does not have this level of understanding, it will not be able to get past the true data dependency described above. Second, it must understand in this *particular* program that the keys for each insertion are distinct. These keys arrive via the `list` data structure, whose elements are produced



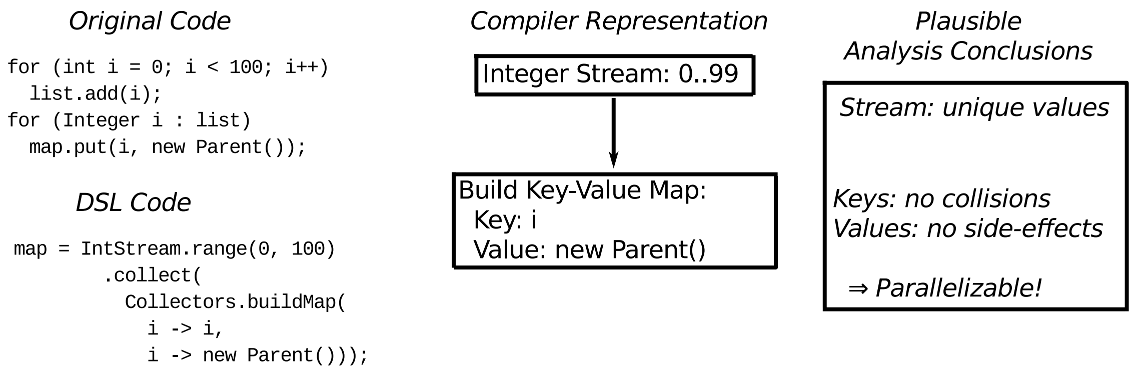
```
for (Integer i : list)
  map.put(i, new Parent());
```

```
for (Iterator it = list.iterator();
     it.hasNext(); ) {
  Integer i = it.next();
  int hashcode = i.hash();
  Parent p = new Parent();
  Entry e = new Entry(i, p);
  e.next = map.slots[hashcode];
  map.slots[hashcode] = e;
}
```

Problem:
Compiler sees data-structure implementation and low-level heap objects

Programmer sees "insert a new object at each distinct key"

Figure 1.2: Compiler view of example program: the compiler does not understand map-insert operations, but instead puzzles through the details of the hash-table update code.



```
for (int i = 0; i < 100; i++)
  list.add(i);
for (Integer i : list)
  map.put(i, new Parent());
```

```
map = IntStream.range(0, 100)
        .collect(
          Collectors.buildMap(
            i -> i,
            i -> new Parent()));
```

Figure 1.3: The first two loops of Fig. 1.1 rewritten using a DSL (here, Java's streaming collections framework). Because operations are expressed at a higher level, a DSL compiler could plausibly analyze the data-structure operations as shown, enabling high-level parallelization optimizations.

by the first loop. The analysis must somehow understand that (i) the data structure at `list` is a list of objects, (ii) the first loop builds the list by appending one element per iteration, and (iii) the same element is never added twice. Then it must use this knowledge when analyzing the map insertion. This level of understanding and analysis is beyond what compilers today are capable of performing.

1.3 A Potential Approach: Domain-Specific Languages

A promising approach today to this specific problem is the *domain-specific language*, or DSL. These languages or programming frameworks enable the programmer to directly write a specification-like description of the problem. This specification occurs at a high level with primitives that are designed for a particular problem domain. DSLs have been designed for many domains and underlying data structures: for example, graphs [69], meshes [39, 22], matrices [86, 9], machine-learning models [105, 111], compiler IR [48], distributed data sets [76, 29, 80], and maps, sets and lists [63]. A closely related approach is to provide a framework in a general-purpose language with DSL-like primitives. One common example is MapReduce [38]: the user fills in behavior while the framework has top-level control and mandates certain restrictions on user-provided functions. These restrictions may place limits on how provided code can depend on or alter global state, for example. Because the domain-level specification in the DSL disallows program behavior outside the bounds of the given primitives, the DSL compiler has significant freedom to choose an optimal implementation strategy.

In the running example of Fig. 1.1, a suitable DSL for expressing the program's computation might contain primitives for key-value map and list data structures, and then primitives to traverse data structures and produce new ones. Fig. 1.3 shows the first two loops of our example expressed with primitives from Java's streaming collections framework [4], which is representative of the constructions that such DSL might allow.[2] A compiler for such a DSL would explicitly understand these primitives, and so could plausibly analyze the operations as follows. First, the stream of keys from the first loop contains no duplicated values, because it is a range of integers. Next, the function provided to the *Build Key-Value Map* operator has no side-effects. Therefore, the *Build Key-Value Map* operator can parallelize its work, choosing data-structure and work-scheduling implementation details as necessary.

DSLs work well when a programmer is developing new code because there is no need to reverse-engineer the meaning of low-level details. Instead, the overall structure of the algorithm can be expressed directly as it is initially designed. This often results in less programmer effort on implementation and debugging by abstracting away irrelevant details. DSLs also work well when an application relies on a small kernel of hot code, because the effort to port or rewrite the kernel into the DSL is relatively minor compared to the large potential benefits.

However, a DSL-based approach suffers from three main drawbacks. The first is that significant effort is often required to port an existing application's logic into a DSL. Furthermore, in such a retrofit, there is significant risk that the new program might not exactly correspond to the original. The programmer has to work carefully to verify that their high-level understanding of the program's operation is correct: for example, the programmer must verify that there is no hidden dependency between parts of the program thought to be independent and self-contained.

The second drawback is that the limited expressive range of a DSL can prevent its use in applications that require an occasional exception to a semantic restriction. For example, a loop that *almost* never has loop-carried dependencies, except for an uncommon invariant-maintenance code path, might be awkward or

[2]The actual Java framework requires the user to explicitly indicate when a stream operator is parallelizable, and is an ordinary library that does not make use of any special compiler intrinsics or support. We merely use its API here as an example of the sort of primitive that a DSL might provide.

inefficient to express. There may be ways to adapt a parallelization strategy to accommodate the exception, but the DSL often prevents the algorithm from being expressed at all. There is evidence that programmers sometimes work around limitations in DSLs, or else simply under-utilize the DSLs, by wrapping DSL kernels with pre- or post-processing code. For example, Cheung et al. [30] find that some program logic surrounding SQL queries can be moved into the SQL itself, but this requires careful analysis, and not all program logic is eligible.

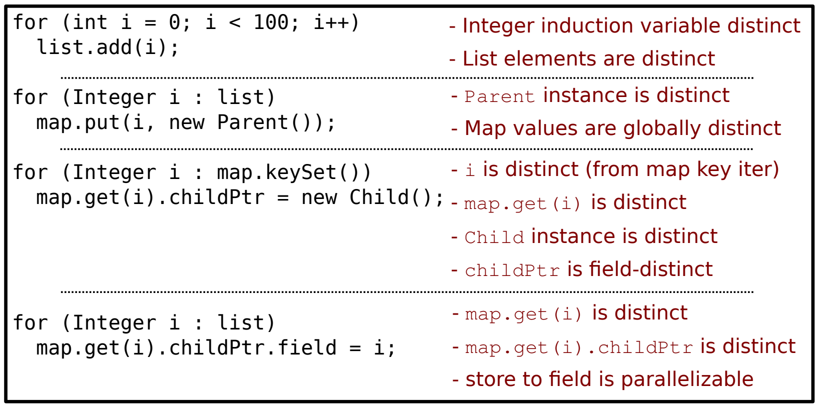
The third drawback is that the DSL is specific to one domain, yet some problems require abstractions from two or more domains. Purpose-built systems exist at these domain intersections. For example, a recent database system, BlazingDB [1], explicitly supports both SQL queries and machine-learning operations directly on in-memory database contents. This combination works particularly well because the in-memory database exposes database contents as numeric data that a machine-learning toolkit's operators can use directly. While it is fortunate that such systems exist for some use-cases, it is significant work to build integrated systems for every possible pairing of domains. Instead, it would be better to have a system that can represent primitives from both domains and compile them into a high-performance transformed program on a shared runtime.

1.4 Our Approach: High-Level Understanding from General-Purpose Languages

For all of the above reasons, we believe that programmers are better-served by a compiler that finds performance in high-level optimizations yet allows them to write code in a general-purpose language. Programmers then could perhaps employ libraries where needed for productivity, but would not be limited by the semantic constraints of a DSL-based approach. Ideally, the system would run an analysis that can identify some high-level structures, such as parallelizable loops, among an otherwise low-level "sea of operations." This should yield good performance in many cases yet remain applicable to many more applications and use-cases than the current state of the art.

Our goal in this thesis is to build such a system, bringing the benefits of DSL-like program understanding and optimization to general-purpose languages. In particular, in this thesis, we focus on optimizing for parallel hardware by *parallelizing loops* in sequential programs. The primitives that our system recognizes, and the analyses that it performs, will be tuned with this initial goal in mind. However, the framework is general, and could serve as the basis of many further code transforms beyond loop parallelization.

Our approach to bridging the gap between general-purpose languages and high-level understanding is twofold. First, we introduce a few *first-class data structures* whose operations are understood by the compiler as intrinsics at the same level as object field or array accesses. The relevant portions of the program and



```
for (int i = 0; i < 100; i++)              - Integer induction variable distinct
  list.add(i);                             - List elements are distinct

for (Integer i : list)                     - Parent instance is distinct
  map.put(i, new Parent());                - Map values are globally distinct

for (Integer i : map.keySet())             - i is distinct (from map key iter)
  map.get(i).childPtr = new Child();       - map.get(i) is distinct
                                           - Child instance is distinct
                                           - childPtr is field-distinct

for (Integer i : list)                     - map.get(i) is distinct
  map.get(i).childPtr.field = i;           - map.get(i).childPtr is distinct
                                           - store to field is parallelizable
```

Figure 1.4: Conclusions derived by distinctness analysis on the example program (Fig. 1.1): the analysis can prove that the second, third, and fourth loops are parallelizable by showing that stored-to heap locations are *distinct*, i.e., different every iteration.

standard library are mapped to these intrinsics. This program representation provides a basis for all further analysis. We find that incorporating just a few data structure types, such as lists and maps, enables native representation of many diverse benchmarks' core data structures. This is because programmers tend to build higher-level data structures and application logic compositionally using these lower-level primitive data structures. Reasoning about these intrinsics rather than their pointer-level implementations grants the compiler significantly enhanced precision that makes our later analyses possible.

Next, we introduce a static analysis, and later a hybrid static-dynamic analysis, to augment alias analysis by using this new information. This new type of analysis, which we call *distinctness analysis*, can derive invariants on data structures such as "each value slot in this key-value map points to a different object." It could then propagate this invariant into a loop that iterates over the map, and as a result, conclude that the map value seen at each iteration does not alias across iterations. This analysis is designed to have enough precision to allow loop parallelization and to enable discovery of other patterns that map to high-level DSL operators: the common thread is to recognize dependencies between different loop iterations or other program elements. Distinctness analysis attains wider applicability than past loop parallelization systems because it does not try to derive *too much* information: for example, it does not compute a closed-form indexing expression that symbolically describes an array access within a loop nest. Rather, it only computes simple loop-relative non-aliasing facts, because parallelization only needs to prove non-aliasing.

Fig. 1.4 shows a series of conclusions by which distinctness analysis may conclude that the last three loops of the example program may be parallelized. The analysis proceeds by deriving a number of invariants on data structures in memory. First, it determines that list elements in `list` are distinct, and then that values in `map` are distinct. It can carry this latter invariant through the heap to the third loop, concluding that the visited `Parent` object is different each iteration. From this and the fresh `Child` allocation, it can infer that `childPtr` is distinct: in other words, no two `Parent` fields point to the same `Child`. This finally enables it to conclude that the stored-to object in the fourth loop is also distinct every iteration. By observing that the destinations of stores are different each iteration, the analysis can determine that the loops are parallelizable.

1.5 Overview of Related Work

Significant past work addresses the loop parallelization problem. A large body of work also aims to enhance compiler knowledge of program behavior to enable better optimization. We summarize various approaches here, and then later discuss these and other areas of related work in more detail in the relevant Chapters that introduce our system.

1.5.1 Parallelization of Regular Loop-nests with Array Accesses

Prior work on the loop parallelization has mainly focused on regular scientific or numeric code (e.g., [59, 66, 36, 18, 45, 78, 84]). Such programs normally consist of loop nests over one or more dimensions, and these loops contain loads and stores to arrays with indices that are linear functions of loop indices. Because the accessed memory locations can be described as simple linear functions, the system can check for potential loop-carried dependencies with precise tests. Such a dependency inhibits parallelization; conversely, if no dependency exist, then every iteration performs independent work, and so iterations may be spread across parallel hardware using a variety of runtime strategies.

Though these previous systems are often effective for scientific and numerical programs, the opportunity for speedup quickly degrades when moving to other application domains. Many programs use data structures other than simple arrays, store data in these data structures in complex patterns without closed-form descriptions, and access and mutate the data in non-linear ways. Thus the past array-based systems are not a complete solution to the auto-parallelization problem. In contrast, our system is able to parallelize some of these difficult-to-analyze loops that nevertheless perform independent work per iteration by incorporating knowledge of data structures and their invariants and the ways in which programs traverse them.

1.5.2 Parallelization using Alias Analysis

Johnson et al. [53] approach the loop parallelization problem by directly resolving the cross-iteration memory aliasing question. Their system contains an ensemble of small, single-purpose analyses, each of which can perform a particular type of deduction. This approach, similarly to ours, recognizes that ordinary compiler alias analysis, which is unaware of loop iterations, is insufficient for parallelization. It thus directly encodes cross-iteration aliasing as a first-class analysis result, as we do. However, their analysis operates at a lower level than ours because it does not recognize first-class data structure values. Rather, it analyzes heap accesses in both the user program and in standard-library data structures at the memory-word level.

Furthermore, their system approaches loop parallelization by posing the aliasing questions needed for loop parallelization directly, in a bottom-up fashion, while our analysis systematically derives distinctness across the program and then evaluates the parallelizability of all loops at once.

1.5.3 Data-Structure Awareness

A number of systems enhance compiler analyses to reason about container data structures in order to enable parallelization. A common use-case is to profile usage patterns and pick an appropriate data structure, or appropriate implementation of a given data structure (e.g., [35, 99, 57, 40, 20, 37, 51]). Two prior lines of work explicitly address parallelization. The Galois system [63, 62] is a framework that provides explicit container data structures for the programmer to use that track operations in order to detect conflicts during speculative parallelization. Like our work, the Galois project observes that knowledge of data-structure API semantics enables greater parallelization flexibility by permitting loop parallelizations that result in different but equivalent final states. In particular, this system is designed to recognize and take advantage of commutative updates. However, the Galois runtime is a library that requires the user to develop the program with library-specific primitives, similar to a DSL. In contrast, we analyze unmodified programs in a general-purpose language, specifically focusing on Java. Wu et al. [115, 114] build a system that reasons about parallelizability of loops that mutate data structures such as hash-tables. While their compiler is aware of the data structure operations and their semantics, as ours is, their system requires insertion of dynamic checks to validate, e.g., that keys inserted into a hash-table do not overlap. In contrast, our system performs static *distinctness* analysis to derive the needed invariants.[3]

1.6 Thesis Statement and Contributions

In this dissertation, we provide evidence for the following thesis:

> *Many programs that were not parallelizable under prior auto-parallelization systems can be safely, efficiently, and effectively parallelized by encoding program operations on data structures as first-class primitives and then analyzing the program's behavior with respect to the heap and those operations on the heap.*

We substantiate this thesis by making the following contributions:

Data-Structure Primitives: First, we introduce a set of compiler primitives, or intrinsics, that enable operation on the fundamental container data structures of *lists, maps and sets as first-class values in the compiler's intermediate representation (IR)*. We show how a small set of semantic models using these primitives can be used to map standard-library implementations of data structures to these

[3]Our Icarus extension also performs dynamic checks, but these strictly improve precision and are not necessary for correctness.

intrinsics during static analysis. We demonstrate that this use of first-class built-in data structures yields significantly increased precision in standard static analyses, and in particular a may-point-to (alias) analysis, compared to a naïve analysis of the original standard library at the implementation level.

Distinctness, an Enhanced Alias Analysis: Next, we introduce *distinctness analysis*, a special type of alias analysis that enhances the results of the baseline may-point-to analysis to enable loop parallelization. Distinctness indicates whether particular variables may alias themselves across iterations of a given loop, whether particular object fields may alias themselves across instances of the object represented by a given heap abstraction, or whether particular map values may alias themselves within a map. We show that distinctness analysis, when combined with use of the data-structure intrinsics introduced above, is able to derive useful non-aliasing invariants on loops in many programs beyond the traditional scientific/numeric realm of auto-parallelization. Our system achieves significant speedup as a result of parallelizing these loops.

Hybrid Static/Dynamic Distinctness Analysis: Finally, we introduce *a hybrid static analysis / dynamic check* framework. This framework extends a static analysis such as distinctness analysis with the ability to use dynamically-verified program properties in its reasoning. The resulting transformed program verifies these properties and falls back to non-transformed code when a check fails. We apply these principles to distinctness analysis, and we show that the result enables parallelization of additional loops with only a few dynamic checks. We describe a runtime scheme to execute these parallelized loops while accounting for possible dynamic-check failure. Unlike most prior parallelization systems that rely on dynamic checks of program invariants, our system is not speculative, hence does not require any buffering or rollback capability. It is also able to take advantage of *partial* parallelizability, wherein a loop's iterations are mostly parallelizable except for a few true dependencies, handled via dynamically-inserted serializations.

1.7 Structure of the Thesis

This thesis will proceed as follows. First, Chapter 2 provides background on several analysis topics on which our contributions rely. The chapter covers the current state of loop parallelization, including approaches for scientific/numeric code with linearly-indexed arrays and generalized approaches based on alias analysis, as well as background on Andersen points-to analysis [12], which is the baseline alias/points-to analysis on which our system is built. It also includes descriptions of techniques for interprocedural and context-sensitive analysis that are used in our system. Chapter 3 demonstrates the need for first-class data-structure values at the compiler IR level by describing how current alias analyses fail to understand code that uses common container data structures, and then introduces our compiler data-structure intrinsics that address this problem. It shows how to extend points-to analysis to account for these data structures, and demonstrates that precision is greatly enhanced as a result. Chapter 4 first describes how a standard alias analysis, even with enhancements for first-class data structures, does not answer the questions necessary to enable loop parallelization because it does not derive aliasing invariants relative to loop iterations. It introduces *distinctness*, a type of aliasing that addresses this need, and describes how distinctness invariants are propagated into the heap (as invariants on heap object fields and map value slots) and back into program variables. It then describes a set of inference rules that determine which loops are parallelizable based on distinctness-analysis results, comprising our system DAEDALUS, and evaluates this system on a number of benchmarks. Chapter 5 introduces ICARUS, our extension of DAEDALUS designed to include dynamic invariant checks to increase analysis precision. It describes how to modify, in a systematic way, the inference rules of the static distinctness analysis in order to propagate "possibility" forward to the parallelization logic and "need" backward to the distinctness analysis, so that only an approximately-minimal number of dynamic checks are inserted to enable the desired loop parallelization. The chapter describes the subtleties of tracking dynamic distinctness and the mechanisms that handle the checks in the parallelization runtime, and evaluates the performance of ICARUS against DAEDALUS and the baseline array-based loop-parallelization system. Chapter 6 describes several promising future research directions and concludes. Finally, Appendices A and B include full definitions and soundness proofs for DAEDALUS and ICARUS, respectively.

Chapter 2

Background: Loop Parallelization and Alias Analysis

There once was a static analysis
That suffered from painful paralysis.
It traversed all the loops
And jumped through some hoops
But failed to prove lack of an alias!

Before we can introduce the key aspects of our system, we must provide some background relevant both to the loop parallelization problem specifically and to alias analysis more generally. This chapter serves as background useful to understand the context and assumed baseline of later chapters. We first describe *loop-nest parallelization analysis* (§2.1) and a standard approach that works by analyzing linearly-indexed array accesses (§2.2). We then provide an introduction to *alias analysis* (§2.3). We describe Andersen points-to analysis, a standard alias-analysis approach (§2.4). We extend this analysis beyond single procedures to whole-program, or *interprocedural*, analysis (§2.5), in a context-sensitive way. We then provide a foundation for the analysis descriptions in the rest of this dissertation by providing notation for inference rules and program representations and giving a simple formalization of Andersen points-to analysis in this notation (§2.6). Finally, we discuss some additional related work (§2.7).

2.1 Loop Parallelization

In this thesis, we work toward a solution to the *loop parallelization* problem. Loop parallelization, generally speaking, aims to execute the iterations of a loop in a sequential program using parallel hardware in some way. We say that a loop is parallelizable if this modified version of the program, executing on the parallel hardware, produces identical output (for some definition of identical) to the original. The loop iterations may

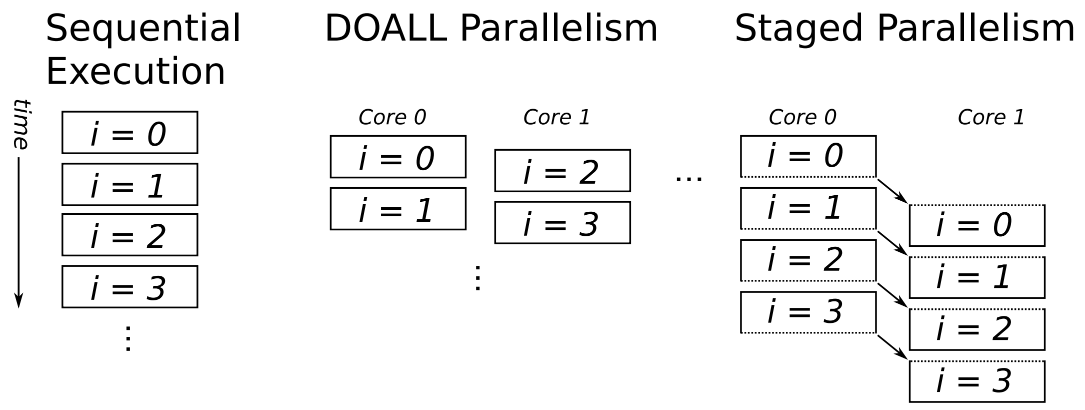


Figure 2.1: Two ways of extracting parallelism from a program loop: executing iterations independently on separate cores (DOALL parallelism) or splitting each iteration into multiple pieces, running each section of the loop body on a different core (staged parallelism).

be distributed across CPU cores to run independently, in classical DOALL parallelism [66], or the iteration body may be split into pieces lengthwise along the execution of a single iteration to form a "pipeline" of stages, attaining parallelism in another dimension [89]. These two forms of parallelism can be combined as well [88]. Other means of parallel work scheduling can be used, too: for example, the dependency structure between iterations may be a partial order because some particular iterations use results from particular previous iterations, but parallelization of such a loop is still possible if the scheduler can respect this dependency. These forms of available loop-related parallelism are summarized in Fig. 2.1.

In this dissertation, we initially focus on cross-iteration DOALL parallelism, and later extend our approach to schedule around true dependencies dynamically. However, fundamentally, the analyses that we introduce will derive general invariants about the program and its heap accesses, and these results could be used to explore other forms of parallelism (on other types of loops, on recursive programs, etc.). We leave such explorations to future work.

2.1.1 Basic Parallelization Requirements

What must be true about a loop for it to be parallelizable? In general, this question reduces to the Halting Problem: we cannot analytically or symbolically determine a program's output in general, and so cannot prove a correspondence between serial and parallel versions in general. As a concrete example of this difficulty, program execution may include arbitrary pointer computations, hence even the basic dataflow between program operations and the dependencies between them are undecidable in general. However, one can define sufficient conditions that will allow us to answer this analysis question for many programs in practice.

The simplest condition that is sufficient to show parallelizability is *complete independence*: if an iteration of the loop body does not interact in any way with any other iteration, then its execution will not be affected by the relative timing of its own execution and other iterations' executions. Then, if iterations can mutate their own separate portions of program state in any order relative to each other, they can be executed in parallel.

Showing this complete independence of each iteration's computation requires proving three properties. The first is simple: the loop body must have no loop-carried dependencies among local variable accesses. This can be tested in a straightforward way if the compiler IR is SSA (static single assignment) [93]: the only way for a definition in the loop body to be used in a subsequent iteration is for it to flow through a ϕ-node in the loop header. This can be directly tested by the analysis.

The second condition is likewise fairly simple: the set of iterations to be executed by the loop must be enumerable prior to the execution of any iteration. In other words, a loop should not have an exit condition that depends on calculations in the loop body. This, too, is straightforward to verify by pattern-matching known loop patterns for consideration. For example, the standard idiom in C-like languages for a loop over a range of integers, namely `for(i = 0; i < N; i++)`, can be easily recognized. As long as the analysis ensures that no other mutations to `i` or `N` occur inside the loop, then it has successfully recognized the pattern and can provide a closed-form description of the iteration indices. Likewise, loops over collection data structures using iterators usually occur in a standard, recognizable form.

The third condition is the most complex, and in fact, is the problem addressed by the bulk of this dissertation: iterations can have no dataflow dependencies through the heap. The analysis shows this by showing that the set of memory addresses accessed by each iteration is disjoint from that of any other, except for addresses that are only read and never written. If this were not the case, i.e., if there were a memory address A that were written by one iteration and read by another later iteration in the original sequential program, then running the iterations out-of-order in parallel could result in a different program output. This would violate our definition of parallelizability.

Systems that parallelize loops have proposed various ways to analyze program memory accesses. We cover two of them here: first, the *loop-nest* model, and second, a model that formulates explicit aliasing questions.

2.2 Parallelizing a Loop Nest with Linear Array Accesses

The simplest type of loop, or nested set of loops, for an auto-parallelization system to analyze is that of a *scientific or numerical kernel* whose only memory accesses are to arrays, such that those accesses are linear or affine functions of the loop indices of the various nested loops. Such a program can be readily analyzed for overlapping iteration memory-access sets because the set of memory locations to be accessed

```
for (i = 0; i < N; i++) {
  a[2*i]     = f(i);
  a[2*i + 1] = g(i);
}
```

Figure 2.2: Simple example program to illustrate parallelization of loops that access arrays with linear indices.

can be summarized with a set of simple linear functions, one per access. An overlap between iterations then corresponds to a solution to an integer linear program formulated in a particular way. Although integer linear programming (ILP) is NP-complete in general, many works have proposed simple tests or frameworks that can solve this problem for common cases.

To make this analysis concrete, consider the program in Fig. 2.2. The loop in this program traverses a one-dimensional space with index variable i. It stores values to two elements of the array a, at indices $2i$ and $2i + 1$ respectively. If we wish to prove that this loop is parallelizable by showing that different iterations' memory accesses cannot overlap, then we need to show that given two iterations i and i' s.t. $i \neq i'$, no pair of two index expressions on a can be equal: that is, that $2i \neq 2i'$, $2i + 1 \neq 2i' + 1$, and $2i \neq 2i' + 1$. This particular case is trivial to see, but other cases become much more complex.

In general, the loop iteration space can be characterized as an n-dimensional vector space $\vec{i} = \langle i_1, i_2, \dots, i_n \rangle$ where each dimension corresponds to one loop in the loop nest. Then, if the loop accesses an m-dimensional array, we can describe the accessed array element $\vec{a}$ in terms of loop iteration $\vec{i}$ as:

$$\vec{a} = M\vec{i} + \vec{c}$$

where M is a matrix representing coefficients of loop index variables in each dimension of the accessed array index, and where $\vec{c}$ is a vector representing the constant offsets. If we have k array accesses to a given array, characterized by $M_1, \dots, M_k$ and $\vec{c}_1, \dots, \vec{c}_k$, we can say that the loop is not parallelizable if there is any integer solution to:

$$M_k\vec{i} + \vec{c}_k = M_{k'}\vec{i'} + \vec{c}_{k'}$$

for any $\vec{i} \neq \vec{i'}$, and for any pair of accesses k and k' (same or different) such that at least one access is a write.

This type of analysis is often called *polyhedral analysis*. This is because the visited loop iteration indices can be seen as defining a polyhedron in n-dimensional space; likewise, the set of visited array elements for each array is a polyhedron in m-dimensional space.

The first systematic test to take a linear indexing function-based approach to the loop dependency analysis problem was in Lamport [66]. The Lamport test works only for single-dimensional iteration spaces and arrays where coefficients are equal among all accesses, but the test itself is very simple: there is a dependency if the difference in constant offsets is divisible by the coefficient. This is sufficient to allow parallelization of the loop in Fig. 2.2. The GCD test, proposed by Banerjee [19], is more powerful as it allows differing coefficients. Many subsequent tests have been proposed [36, 18, 45, 49, 83, 78, 84], including the Delta test [49] and the Omega test [83]. Loop-nest-based reasoning is currently used in production compilers such as LLVM's Polly [50]. We will not describe the details of these works here other than to note that all works in this category are fundamentally based on an analysis of array accesses within loops that have numeric index variables.

The loop-nest model's simplicity affords significant flexibility in tuning. A fruitful line of work leverages the flexibility of the model to enable high-level optimizations. These include loop interchange [113], which rearranges loop nest ordering, as well as tiling or blocking for better cache locality [65], or inserting prefetch operations to hide memory latency [75]. Some works also apply machine learning to explore all of the above dimensions for the best performance [106, 16, 15]. However, while this restricted domain covers a large number of numerical loop kernels, providing high performance on supercomputer-class parallel hardware executing these programs, it is not sufficient to address the increasing need for parallelism in other sorts of programs that use heap data structures other than arrays. To parallelize loops in such programs, we need to adopt a more general framework to the memory-access analysis problem.

2.3 Alias Analysis

We next provide a brief overview of *alias analysis* and the closely related *points-to analysis*. Alias analysis is the basis of another approach to the loop parallelization problem, more general than loop-nest array-indexing analysis but also more computationally expensive.

The analysis question that is answered by an alias analysis is: given pointer variables x and y in a program, can any value assigned to x have the same pointer value as any value assigned to y in some execution?[1] It is clear that this kind of analysis can be used to determine whether loop iterations are independent: an overlap of memory accesses across iterations is a pointer alias, and so answering some set of aliasing questions in the negative should be sufficient to show parallelizability.

Points-to analysis is closely related to alias analysis: it computes essentially the same information, only represented in a different way. A points-to analysis categorizes all objects that might exist at runtime into *heap abstractions* and then computes the set of heap abstractions to which each pointer variable might refer.

[1]This is a *may-analysis*, i.e., it computes may-alias relations. There are also must-analyses that compute when a particular value assigned to x *must* be the same as a particular value assigned to y in *all* executions.

(Heap abstractions are frequently defined to represent all objects allocated by a particular statement, or a particular statement in a particular runtime context, but this design choice is arbitrary and affects only analysis precision, not correctness.) Each heap abstraction, representing an object of a certain type, also has a points-to set for each pointer-typed object field. The resulting information is sometimes called the *points-to graph*: program variables and heap abstractions are nodes and points-to relations are edges. One can answer an aliasing question using a points-to analysis result simply by testing whether the points-to sets of the variables in question have a non-empty intersection.

Several prior works have used this general alias-analysis framing to address the loop parallelization problem. Recently, Johnson et al. [53] proposed an ensemble of several smaller alias analyses that formulate and collaborate on dependency queries, with an explicit notion of cross-iteration dependencies. Earlier, Wu et al. [114] based their approach on a refined points-to analysis, though with aspects of the array-based approach as well: their analysis qualifies points-to conclusions further by indicating which concrete object is pointed-to, if this can be derived. While we take a similar approach with respect to an aliasing-based problem framing, our work in this dissertation (i) incorporates data-structure awareness and (ii) introduces a novel form of aliasing that is derived by a systematic, whole-program analysis, unlike these earlier works.

2.4 Andersen Points-to Analysis

We now briefly describe a standard points-to analysis that serves as the basis of many other analyses, including our own analysis introduced in Chapter 4.

There are two efficient, practical forms of alias analysis in frequent use: Andersen analysis [12] and Steensgaard analysis [103]. Of the two, Andersen analysis is more precise, at the cost of some additional computation. We build our system on Andersen analysis, which we now describe.

The key idea of Andersen analysis is to create *set constraints* among points-to sets based on program statements, without regard to the statements' order, and then solve these constraints. Because the constraints always specify that one points-to set is included in another, the analysis problem must have a solution. In fact, it is monotonic: adding another program statement can only ever grow points-to sets.

Fig. 2.3 shows the rules by which Andersen analysis generates set constraints and an application of this analysis to a simple program. The rules can be understood with a simple intuition: the analysis propagates heap abstractions along program dataflow just as the rules of the original language semantics propagate concrete pointer values. Thus, an assignment propagates the contents of the source's points-to set to that of the destination. (SSA ϕ-nodes are analyzed as multiple assignments, one per source). A load propagates abstractions from the given field in all possible referenced objects to the destination. Finally, a store propagates abstractions to the given field in all possible referenced objects. The points-to sets are initially

Andersen Points-to Analysis

pts(x) *denotes points-to set of* x
pts(A.f) denotes points-to set of field f
of abstraction A
A_1 *denotes a heap abstraction*
(all objects allocated at statement 1)

Assignment	
[x := y]	pts(y) ⊆ pts(x)
Loads	
[x := y.f]	∀ A ∈ pts(y). pts(A.f) ⊆ pts(x)
Stores	
[x.f := y]	∀ A ∈ pts(x). pts(y) ⊆ pts(A.f)
Allocations	
[x := new T]	A_n ∈ pts(x)

Example Program

x = new T();	pts(x) = { T_1 }
y = new T();	pts(y) = { T_2 }
w = new U();	pts(w) = { U_1 }
u = new U();	pts(u) = { U_2 }
v = new U();	pts(v) = { U_3 }
x.f = w;	pts(T_1.f) = { U_1, U_2 }
x.f = u;	
y.f = v;	pts(T_2.f) = { U_3 }
z = x;	
z = y;	pts(z) = { T_1, T_2 }
a = z.f;	pts(a) = { U_1, U_2, U_3 }

Points-to Graph

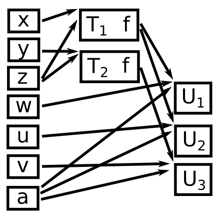


Figure 2.3: Andersen points-to analysis definition and example. Here we show (a) the rules that generate set constraints, (b) a simple example program to analyze, and (c) the resulting points-to sets and a graph representation of them.

populated with abstractions at their points of origin: here, abstractions categorize objects by allocation site, so allocation statements each populate their destination points-to set with one abstraction.

We will describe a common inference-rule notation for these rules in §2.6.2 below. First, we enhance the analysis with some necessary features.

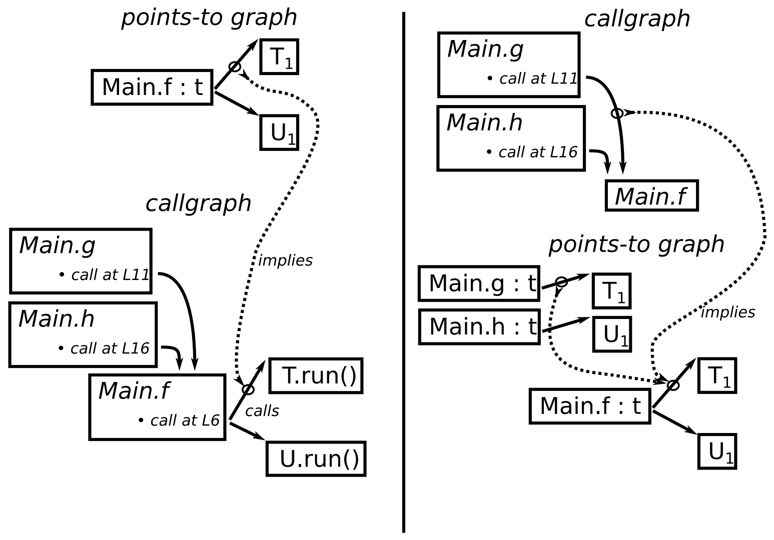
2.5 Interprocedural Analysis

Analyses are commonly described in a simplified form relative to the implementations of real program static analysis frameworks. One such simplification is the assumption that the whole program is *one sequence of statements*, with no function or method calls or returns. To analyze a program in any programming language with function or method calls, it is necessary to extend the analysis. All of the techniques that we describe below are well-known; we simply provide a summary here for the reader's benefit and to explicitly specify the baseline analysis.

Extension of an analysis to a program of multiple functions or methods requires several steps. First, the analysis must be able to name the program points and variables in different functions. This is just a matter of prefixing all identifiers with method names. The analysis must then reason about *interprocedural control flow edges* (calls and returns) in the same way that it reasons about intraprocedural control flow edges. This requires the *callgraph* to be available: the callgraph contains an edge from each invocation site (call statement) to each method that could be called. Finally, the analysis is usually extended with *contexts*: a single method may be called from multiple locations, with very different program states and very different parameter values at each callsite. If all invocations were analyzed as one merged problem, the analysis would lose significant precision. Hence every method, statement and variable name is augmented with a context,

```
1  class T { ... }
2  class U extends T { ... }
3
4  class Main {
5    void f(T t) {
6      t.run();
7    }
8
9    void g() {
10     T t = new T();
11     f(t);
12   }
13
14   void h() {
15     T t = new U();
16     f(t);
17   }
18 }
```

(a) program with multiple functions



(b) points-to graph influences callgraph (c) callgraph influences points-to graph

Figure 2.4: Extending an analysis to support programs with multiple functions or methods (*interprocedural analysis*) requires construction of a callgraph simultaneously with points-to analysis: (i) points-to resolution creates new possible callgraph edges, and (ii) dataflow across callgraph edges can grow points-to sets.

and a callgraph edge links to a clone of the called method with a particular context.

2.5.1 Call-Graph Construction and Analysis Extension

In order to analyze a program with multiple functions or methods,[2] the points-to analysis constructs a *callgraph* describing invocations from call statements to invoked functions or methods. Callgraph edges are added initially for any method call that can be resolved directly (for example, calls to static methods). Then, for virtual method calls, the callgraph is constructed with the help of the points-to graph using a straightforward rule: a callgraph edge is added from an invocation statement to the invoked method on each class (type) to which the method call's object variable may point.

Now, as the callgraph is built, it implies dataflow connections between methods: whenever a call statement might invoke a method, there is a potential assignment from the call arguments passed to that statement to the formal parameters of the method. Likewise, there is a potential assignment from the return value of the method to the returned value of the call statement. As a call statement might be linked to multiple potential target method implementations, and a method might be linked to multiple potential callers, these call-dataflow assignments are analyzed as virtual ϕ-nodes, merging all possible values.

Because the callgraph construction depends on the points-to graph, and the points-to graph will grow as a result of the additional assignments created by callgraph edges, these two analyses are mutually dependent,

[2]In this thesis, as we focus on programs in an object-oriented language, Java, we will say *method*. However, nothing in interprocedural analysis requires an object-oriented framework. In fact, in a simpler language such as C, very few calls need the help of a points-to analysis to resolve, as most calls are direct. Only calls through function pointers need points-to analysis to resolve.

and must be run simultaneously until both converge. Fig. 2.4 demonstrates this dependence in more detail using an example program in Java. For the given example, (i) the fact that `t` in method `f()` on class `Main` can point to either a `T` or `U` instance (the analysis has one heap abstraction of each type) implies that the call to the `run()` method on this variable might resolve to the implementation either on class `T` or class `U`. Going in the other direction, the fact that `g()` on class `Main` calls `f()` implies that all contents of the points-to set for the argument variable passed to this call are propagated to the points-to set of the formal parameter on `f()`.

2.5.2 Context Sensitivity

Note that in the analysis of the example program in Fig. 2.4, the points-to set of `t` in method `f()` contains both the heap abstraction of type `T` for allocations at line 10, and the heap abstraction of type `U` for allocations at line 15. From this point forward, all analysis of `f()` is reasoning about a *merged* version of the method, combining possibilities that occur only when called from the callsite at line 11 or line 16, respectively. This imprecise merging can propagate further as well: for example, when `f()` then calls `run()`, which is resolved to the method on either class `T` or `U`, *both* possible abstractions will be passed as the `this` parameter to both `run()` implementations. Of course, this does not happen in practice: in this program, `T.run()` is never invoked with an instance of `U` as `this`, or vice versa.

The solution to this problem is to analyze two different *versions* of the method, one embodying each possible scenario. This is known as *context-sensitive analysis*, and can take many forms. The simplest is to analyze the method separately for each callsite in the program: thus, in our example, we would analyze *f() called from line 11* and *f() called from line 16* as separate methods, with separate variables having separate points-to sets, and so on. This would be sufficient to resolve the imprecision described above. More precision can be attained by using a *call-string* instead, i.e., the caller of the caller, and so on, up to a fixed depth. Fig. 2.5 illustrates this example concretely.

Later in this thesis, we make use of 1-object sensitivity [73], which uses the heap abstraction of the `this` object (method receiver) as a context instead. Past work has shown that this is quite effective at achieving high precision with relatively low cost when analyzing object-oriented programs.

2.6 Program Analysis Definitions

Finally, before proceeding to our contributions, we must introduce some notation. So far, we have described points-to analyses and context sensitivity somewhat informally, using examples in Java and textual descriptions of analysis rules. Here, we define our notation for program statements, provide an overview of the common inference-rule notation, and define Andersen points-to analysis in this notation.

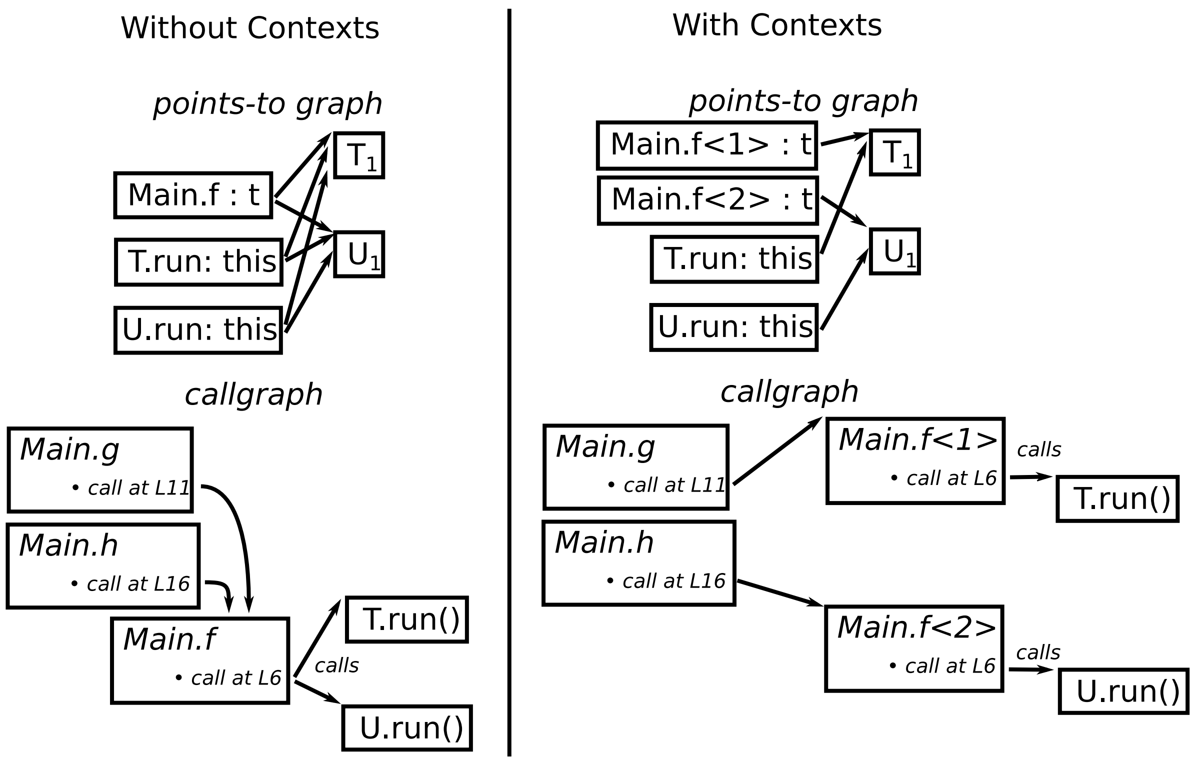


Figure 2.5: Context-sensitive interprocedural analysis computes its results separately for each instance of a method with context. Here, callsite context sensitivity (depth 1) is sufficient to resolve the imprecision seen in Fig. 2.4.

Intermediate Representation (IR), Simplified

x := new T	*allocate new instance*
x := y	*assignment*
x := phi(y, z, ...)	*phi-node*
x.f := y	*store instance field*
x := y.f	*load instance field*
x := y OP z	*binary operator*
x := call obj.f(y, z, ...)	*method call*
return x	*method return*
goto label	*unconditional branch*
if x goto label	*conditional branch*

Notation: Statements and Points-to Sets

$[x := \dots]_i$	*statement i (global index)*
pts(x)	*points-to set of variable x*
A_i	*heap abstraction for object allocated at statement i*
pts(A.f)	*points-to set of field f on heap abstraction A*

Andersen Analysis Inference Rules

$$\frac{[x := y] \quad A \in pts(y)}{A \in pts(x)} \quad \text{[PTAssign]}$$

$$\frac{[x := \varphi(\dots, y, \dots)] \quad A \in pts(y)}{A \in pts(x)} \quad \text{[PTAssignPhi]}$$

$$\frac{[x := \text{new } T]_i}{A_i \in pts(x)} \quad \text{[PTAlloc]}$$

$$\frac{[x := y.f] \quad A \in pts(y) \quad B \in pts(A.f)}{B \in pts(x)} \quad \text{[PTFieldLoad]}$$

$$\frac{[x.f := y] \quad A \in pts(x) \quad B \in pts(y)}{B \in pts(A.f)} \quad \text{[PTFieldStore]}$$

Figure 2.6: Basic program intermediate representation (IR) on which analyses in this dissertation will operate, slightly simplified, and a formalization of the Andersen points-to analysis, demonstrating inference-rule notation and the analysis of this IR. Note that the IR here excludes array operations and exception handling for simplicity, and the analysis excludes interprocedural and context-sensitive details.

2.6.1 Program Intermediate Representation (IR)

Fig. 2.6 defines a number of statement types in a simple intermediate representation.[3] The IR is in static single assignment (SSA) form [93] and provides field loads and stores, object allocation, operators on primitive data types, conditional and unconditional control flow, and method invocation.

Our analysis definitions assume object-oriented programs.[4] These programs have classes with methods and with fields. The program heap consists of objects, each either a scalar instance of a class or an array of primitives or object pointers. Object pointers may be held in local variables, fields, or array slots.

2.6.2 Inference Rules and Andersen Points-to Analysis

We provide a more formal definition of Andersen points-to analysis on the right side of Fig. 2.6 in a standard inference-rule notation. An inference rule, consisting of antecedents above a horizontal line and consequents below it, and usually a label for the rule to the side of the line, indicates that when the conditions above the rule are true, the fact or facts below the rule can be inferred. Analyses can be written as a set of inference rules that are applied in turn, gradually building up a body of facts or *judgments* that describe the program.

The rules for the points-to analysis correspond more or less directly to the specification earlier in Fig. 2.3: each set inclusion constraint is implemented with a rule that transfers a single element from a source points-to set to a destination points-to set under some additional conditions.

In this dissertation, we will specify our analyses using inference rules of this form. This formalization is convenient for soundness proofs and also translates directly to our implementation, which is developed in the Datalog logic programming language.

2.7 Other Related Work: Alternate Approaches

Above, we introduced the loop-nest-based approach to loop parallelization, and then the aliasing-based approach, setting the stage for our contributions on top of the latter. However, for completeness, we will now briefly describe several other approaches to static analysis for loop parallelization before returning to a description of our analysis.

2.7.1 Program Dependence Graphs

Many static-analysis works use Program Dependence Graphs (PDGs) [61, 77], which enable reasoning about dependencies across the program. The key idea in a PDG-based system is to represent the entire program's dataflow, interprocedurally as well as intraprocedurally, as a graph so that program transform conditions

[3]This IR is simplified for expository purposes; the true IR also contains operations on arrays, operations for throwing and handling exceptions, and static and non-virtual method invocations, which we omit.

[4]Note, however, that nothing in our analyses is fundamentally dependent on this assumption; they could easily be adapted to other language models.

can be expressed more simply. Loop parallelization simply requires finding strongly-connected components (SCCs) in a loop body, whose cycles represent loop-carried dependencies, and ensuring that dynamic instances of these SCCs in loop iterations are serialized. (If no cycle exists in the loop body's PDG, then the entire body can be parallelized.) For example, Decoupled Software Pipelining (DSWP) [89, 88] leverages a PDG to decompose loops into "stages," some of which are fully parallelizable and others of which execute sequentially due to loop-carried dependencies. Parallelism is thus found in two different dimensions: across iterations and along each iteration. The Paralax compiler [110] likewise uses a PDG: by requesting some programmer help where the static analysis cannot resolve aliasing, the compiler resolves the PDG accurately enough to parallelize arbitrary C programs.

PDG-based approaches simplify various aspects of program transforms, such as the decoupling of cyclically-dependent subsets of loop iterations as performed by DSWP. However, the representation is somewhat more explicit than the set of loop iteration-relative aliasing invariants that we derive, and hence could be costlier to compute and/or lose precision on more complex programs.

2.7.2 Shape Analysis

A large body of *shape analysis* work exists to compute summaries or symbolic descriptions of a program's heap [55, 46, 112, 21]. In a closely related vein, many works attempt to find program invariants, either explicitly relating to heap-based data structures, or in general on program state. This work ranges in analysis effort and precision from coarse-grained, quickly-computed summaries, such as whether a data structure is cyclic, all the way to systems that use SMT solvers or theorem provers to derive and verify invariants.

Ghiya et al. [46], in a representative early work on shape analysis, specify an interprocedural dataflow analysis that tracks relations between heap pointers. This analysis determines whether access paths may exist between objects at local variables, and then categorizes the sorts of data structures that may exist by *shape*: an acyclic tree, an acyclic DAG, or a cyclic graph. This information can then be used for further analyses. For example, Ghiya et al. in a later work [47] describe in detail how to use points-to analysis to find parallelizable program regions and assume an analysis like the above to improve precision of aliasing information. Because shape analysis can resolve, e.g., that two linked lists are disjoint, as shown in one example in [47], it can help to find additional parallelism. Other works use shape analysis as part of the overall toolkit that constitutes a well-engineered dependence analysis: for example, the Paralax compiler [110] uses Data Structure Analysis [67], a simplified form of shape analysis that improves aliasing precision.

As an example of a more sophisticated and more expensive approach, Berdine et al. [21] describe a system that constructs possibly recursive predicates describing pointer relationships of data structures. The recursive aspect of this system enables repeating data structures such as linked lists to be concisely described. The predicate language is powerful enough to describe many different data structures; however, the derivation requires sophisticated heuristics to correctly derive useful predicates.

Though powerful, using shape analysis to derive precise-enough conclusions for useful loop parallelization is usually too costly. This is because the analysis attempts to derive precise invariants, and these invariants are generally more specific than is needed just to show that loop iterations are commutative.

2.7.3 Separation Logic

A large number of heap-analysis systems are built on Separation Logic [91], a type of logic that enables expression of surprisingly powerful heap predicates. The key idea is to describe pieces of the heap that must not alias, and join these pieces with an operator ($*$) indicating this separation. This allows *local* or *modular* reasoning about particular heap objects with the guarantee that no other part of the program will interfere, which is the approach that most human programmers intuitively take.

While all of these heap-analysis systems are capable of deriving useful heap descriptions of some form, they expend effort to reverse-engineer the *implementations* of data structures such as lists and trees. As we argue in this thesis, this effort may be better spent on analyzing the user program's semantics by incorporating these low-level building blocks directly into the program representation.

2.7.4 Program Invariants

A number of systems derive program invariants more generally, such as relationships between integer variables that can be described with a formula, or relationships between pointer variables. A representative example is the Daikon system [43, 42], which makes dynamic observations of program execution and then proposes possible static invariants. Such a system can be powerful as a complement to a heap-specific analysis. However, these systems are usually more general than our focused approach to deriving particular forms of heap invariants on data structures and pointer variables. Deriving the needed invariants by observation or by proof from first principles would be significantly less efficient than our parallelization-specific analysis.

2.8 Chapter Summary

In this chapter, we described the *loop parallelization problem* in more detail, defining what it means to parallelize a loop and exploring how a program analysis might prove the necessary conditions for such parallelization. We provided a brief overview of the standard loop-nest approach, in which the analysis proves a loop to be parallelizable by showing that a linear equation has no integer solutions. Because this analysis is not widely applicable outside the domain of scientific or numerical programs that operate on arrays, we then described how a more general alias analysis may be used to show parallelizability. We described in some detail the standard Andersen alias analysis. Finally, we provided necessary notation for program IR and inference rules, both necessary to understand the remainder of this dissertation.

Chapter 3

Data-Structure Awareness with Semantic Models

A List, a Set and a Map
Once had here a bit of a flap
Said List: order matters –
Which left Set a-scatters
Said he, "best get sorted, old chap!"

In this chapter, we describe the first contribution of this dissertation in detail: the application of *first-class semantic data structure knowledge* in the compiler to enable more precise points-to analysis of non-numeric/scientific programs, i.e., those with heap data structures aside from arrays.

3.1 The Problem: Analysis of Common Data Structures is Imprecise

The fundamental problem that first-class data structure operations address is the imprecision that arises when points-to analysis and loop parallelization analysis encounter data-structure implementations. There are two aspects to this imprecision: (i) pointer values may be *unnecessarily merged*, so that points-to sets are larger than they might otherwise be, and (ii) the *dependencies* between the operations and their *commutativity* when operating on a common data structure cannot be directly analyzed, which ultimately prevents parallelization of many loops. We now study these problems further.

3.1.1 Problem 1: Imprecise Pointer Dataflow Through Data Structures

The first major issue with a data-structure-unaware approach is that a points-to analysis produces unnecessarily imprecise results when analyzing many common data structures. Let us consider the program in Fig. 3.1. The program inserts two different types of objects into `map`, a `HashMap`, which is a hashtable-based

Original Code

```
HashMap map;
for (Object key1 : keys1)
  map.put(key1, new Obj1());
for (Object key2 : keys2)
  map.put(key2, new Obj2());
Object val = map.get(key1);
```

Analyzed Code

```
for (Iterator it = keys1.iterator();
     it.hasNext(); ) {
  Object key1 = it.next();
  int hashcode = key1.hash();
  Obj1 value = new Obj1();
  Entry e = new Entry(key1, value);
  e.next = map.slots[hashcode];
  map.slots[hashcode] = e;
}
// ...
```

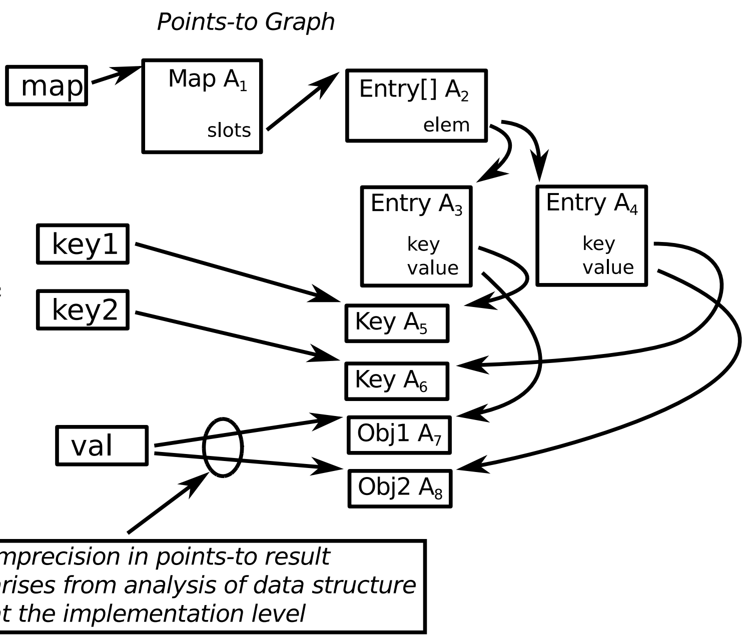


Figure 3.1: Points-to analysis of data structure operations at the implementation level, without awareness of the higher-level data structure semantics, can lead to imprecision through undesired merging of heap abstractions.

key-value map, indexed by keys from two different lists. It then fetches the object at one particular key. (Assume that `key1` is an object from the `keys1` list.) We make several observations. First, as we noted in Chapter 1, the analysis must consider the *implementation* of data structure methods, which contributes to analysis cost, as compared to a hypothetical analysis that directly understands the notion of a key-value map insertion. Second, and more importantly, the points-to graph has undesirable *merging*: any access to the hash-table traverses abstraction A_2 for the `Entry[]` array, indicating only that the array points to two key-value entry abstractions (A_3 and A_4) overall. Thus, for example, the `Map.get()` call assigning its result to `val` will produce a points-to set for `val` containing both A_7 and A_8. Ideally, instead, we would like the analysis to conclude that `val` can only point to abstraction A_7, i.e., an `Obj1` instance.

The imprecision in this analysis arises from the inability of the heap representation to describe the indexing relationship between keys and particular objects. This relationship manifests itself concretely both in the hash-code computation and indexing, and in the hash-table bucket search loop (not shown) that compares keys until it finds a match.

3.1.2 Problem 2: Semantic Dependencies and Commutativity

The other major imprecision that occurs when analyzing a program in a data-structure-unaware way is that the analysis has no knowledge of the dependencies or the allowable reordering, or commutativity, of the operations.

Consider again the program in Fig. 3.1. Assume for a moment that no key appears more than once in the `keys1` or `keys2` list. If this is the case, then both of the loops in the program are parallelizable w.r.t. the definition requiring only an equivalent final API-visible state, as long as the `Map.put()` calls are wrapped in a lock: it does not matter in what order these operations occur, because no operation overwrites the result of another operation.

However, a simple analysis that observes the hash-table implementation will have a very difficult time coming to this conclusion. The primary issue is that there *may actually be a true dependency* at the memory word level: two keys may hash to the same hash-table slot, in which case the second insertion will read and then overwrite the bucket-chain head pointer in `slots[hash]`. In order to disregard this, the analysis has two options. It could prove from first principles that performing the insertions in either order will place both key-value pairs in the hash bucket, and that the hash-table lookup will return the same value irrespective of this order. However, this is unlikely to be possible in an analysis with reasonable cost. (Some past work has taken this approach [92, 10], and Aleen and Clark address analysis cost with a randomized approach to symbolic execution [10], but the approach remains limited.) Instead, the analysis could reason about *built-in intrinsics* for key-value map operations, designed to leverage this commutativity by modeling API-level semantics directly.

3.2 Our Approach: First-Class Data Structures

To address both of the problems described above, we encode *data-structure operations and their semantics* in the IR itself. Thus, rather than invoking an ordinary method that implements, e.g., hash-table insertion, the program can use a `mapput` IR statement directly, passing a *first-class map value* as an operand.

We choose to represent two basic *container data types* as first-class values: key-value maps and lists. A third type, a set of elements, can be built from the key-value map. These few basic container types compose to represent a surprisingly large number of common program data structures, because programmers often use container data types to build hierarchical heap structures in application-specific ways.

In order to enable use of these primitives where appropriate in existing programs, we build support in our system for *semantic models* that override standard-library classes with implementations that use the built-in types. These models are not meant to be compiled or executed, but rather, are replacements with equivalent semantics that can be used for static analysis. Our system's goal is to provide the building blocks that are useful to these models in specifying the behavior of library APIs, while simultaneously designing the abstractions so that tractable levels of static analysis can produce useful results.

The new data-structure operations are summarized in Fig. 3.2 and are described in more detail in the following sections. The two new types of object, map and list, live alongside ordinary class instances, and

Data-Structure Aware IR Operations

Key-Value Maps (§3.3.1)

m := mapnew	*allocate new map*
mapclear m	*clear map*
mapput m, k, v	*insert by key*
v := mapget m, k	*look up by key*
v := mapremove m, k	*remove at key*
x := mapprobe m, k	*test key presence*
x := maplength m	*return map length*
o := equivclass v	*return equiv class*

(Sets are implemented as maps (§3.3.2))

Lists (§3.3.3)

l := listnew	*allocate new list*
listclear l	*clear list*
listput l, idx, v	*write at index*
v := listget l, idx	*read at index*
listinsertidx l, idx, v	*insert new elt at idx*
listremoveidx l, idx, v	*remove elt at idx*
listappend l, v	*append element*
listprepend l, v	*prepend element*
v := listpopfront l	*pop first element*
v := listpopback l	*pop last element*

Iterators (§3.3.4)

x := iterhasnext i	*another elt in seq?*
x := iternext i	*get next elt*
x := iterremove i	*remove elt*
i := mapkeyiter m	*get iter over keys*
i := listiter l	*get iter over elts*

Virtual Values (§3.5)

v := newvirtval	*create a virtual value*
vvread v	*read the virtual value*
vvwrite v	*write the virtual value*
vvcommwrite v	*commutatively write the virtual value*

Figure 3.2: Summary of IR operations for first-class data structure support.

are held by reference just as ordinary objects are. Their contents are accessible with the operations defined here, but they are otherwise opaque. When first-class data structure support is enabled, our system also rewrites arrays as lists, for analysis purposes.

Support for these container data types in analyses follows a basic insight: these containers are closely analogous to ordinary heap objects with fields, except that the definition of *field* depends on the type. An ordinary object has field values that are uniquely identified by a static set of constant field names. In contrast, a key-value map has value fields that are identified by keys that are themselves objects. Finally, a list has value fields that are identified by integer indices. A points-to or alias analysis can handle all three types of heap objects with this insight simply by replacing the use of the static field identifier with the appropriate key-object abstraction, as we will describe in §3.3.1 below.

3.3 Improving Points-to Precision with First-Class Data Structures

We now introduce the primitives for first-class data structures alongside the extensions to points-to analysis that make use of this additional semantic information to more precisely analyze the program. This analysis extension addresses Problem 1, imprecise points-to analysis of data structures, described in §3.1.1 above.

3.3.1 Key-Value Maps

The most important first-class data structure is the *key-value map*. This primitive enables implementation of higher-level key-value map APIs, but can also be used to implement the semantics of lists and sets for analysis purposes, as we will see later.

A map contains key-value pairs consisting of arbitrary objects. The map supports access to the value slot indexed by a given key. The supplied primitives are minimal, but sufficient: the program can insert and remove key-value pairs, test for the presence of a key, look up the value at a key, and fetch the total key-value pair count. A summary of these IR operators and pseudocode representing their semantics is presented in the left half of Fig. 3.3.

IR operation	Pseudocode	Points-to Analysis
m := mapnew	*m := {}*	$\dfrac{[m := \text{mapnew}]_i}{A_i \in pts(m)}$ [PTMapNew]
mapclear m	*m := {}*	
mapput m, k, v	*m[k] := v*	$\dfrac{[\text{mapput } m, k, v] \quad M \in pts(m) \quad K \in pts(k) \quad V \in pts(v)}{V \in pts(M.K)}$ [PTMapPut]
v := mapget m, k	*v := m[k]*	$\dfrac{[v := \text{mapget } m, k] \quad M \in pts(m) \quad K \in pts(k) \quad V \in pts(M.K)}{V \in pts(v)}$ [PTMapGet]
v := mapremove m, k	*v := m[k]; del m[k]*	
x := mapprobe m, k	*x := k ∈ m*	
x := maplength m	*x := | keys(m) |*	

Key

Pseudocode

{}	empty map
m[k]	value in m at k
del m[k]	remove key k
k ∈ m	k present in m?
keys(m)	set of keys in m

Inference Rules

$[m := \ldots]_i$	statement i
pts(v)	points-to set of var v
pts(M.K)	points-to set of M indexed by K

Figure 3.3: Summary of IR operations for first-class maps, including inference rules for extension to Andersen points-to analysis.

Points-to Analysis of Maps

We next extend points-to analysis to support maps as well as ordinary heap objects. Map allocations create *map heap abstractions*, analogous to ordinary object heap abstractions. While the latter contain one points-to set per object field, map abstractions contain one points-to set *per key abstraction*: i.e., fields are replaced by abstractions that represent keys. This follows the basic insight described above: points-to sets are tracked for "fields" indexed by *heap abstractions* as keys as well as constant field names, and the new inference rules are straightforward extensions of the existing rules for loads and stores.

The right side of Fig. 3.3 provides these inference rules. The rules extend the basic Andersen points-to analysis [12] as specified in Fig. 2.3. In brief, a store to a map adds elements to the points-to set identified by every relevant map abstraction and every relevant key abstraction, and a load from a map transfers elements from these same points-to sets to the load result.

The immediate result of this extension is enhanced points-to precision when a map is indexed by multiple heap abstractions: the analysis can distinguish such keys. This contrasts with the analysis that perceives the hash-table implementation as a black box, indexing an array by some unknown integer and performing

Original Code

```
HashMap map;
for (Object key1 : keys1)
  map.put(key1, new Obj1());
for (Object key2 : keys2)
  map.put(key2, new Obj2());
Object val = map.get(key1);
```

Analyzed Code

```
for (Object key1 : keys1) {
  mapput map, key1, new Obj1;
}
for (Object key2 : keys2) {
  mapput map, key2, new Obj2;
}
val = mapget map, key1;
```

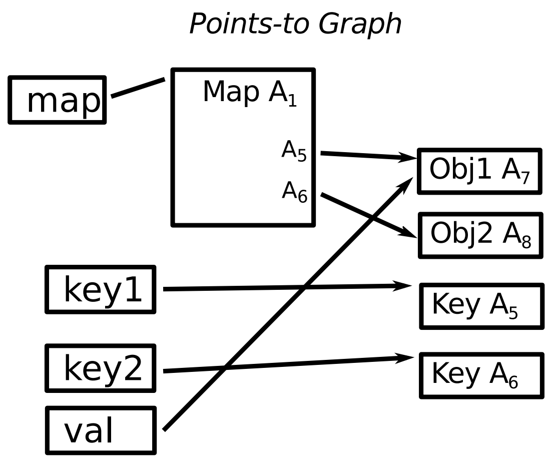


Figure 3.4: Example program and points-to graph of Fig. 3.1, revised with first-class map operators and map heap abstractions.

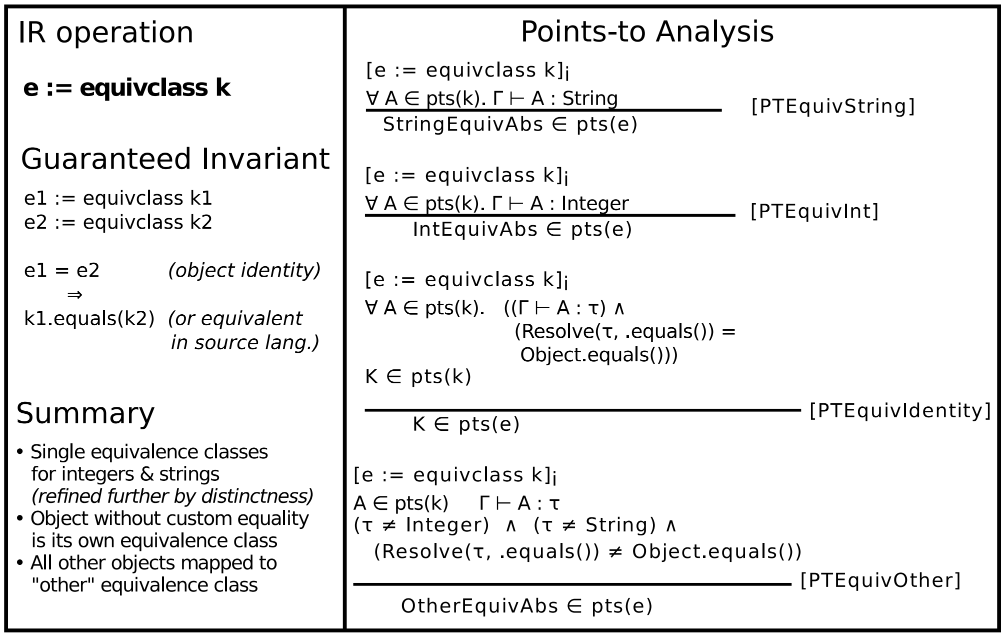
some control flow. One could understand the former from the latter by seeing that the array index is a hash value, and that the control flow is branching on the result of a key comparison, but existing analyses do not do this.

Returning to the example program of Fig. 3.1, we show an updated points-to graph in Fig. 3.4. Note first of all that the points-to graph and the IR are both much simpler: the map abstraction and its operations are both built-in concepts. Importantly, this analysis extension is able to resolve the value in `val`, fetched from the map via a particular key, to have a more precise points-to set than before.

Object Equality and Map Indexing Semantics

So far, we have described a map that indexes its value slots by *pointer value*, i.e., object identity, rather than by any notion of user-defined key equality. This differs from, e.g., the semantics of standard map containers in the Java and C++ standard libraries.

In order to build semantic models for the standard libraries, we must model the usage of these higher-level equality operators, such as the `.equals()` method on Java objects. However, incorporating such semantics directly into the points-to analysis produces significant unnecessary complexity by blending multiple layers of abstraction. Rather, it is much more natural for a points-to analysis, whose sole concern is object identity, to analyze maps that are indexed by object identity. We thus provide another operator that enables modeling of the higher-level equality.



IR operation

e := equivclass k

Guaranteed Invariant

e1 := equivclass k1
e2 := equivclass k2

e1 = e2 *(object identity)*
⇒
k1.equals(k2) *(or equivalent in source lang.)*

Summary

- Single equivalence classes for integers & strings *(refined further by distinctness)*
- Object without custom equality is its own equivalence class
- All other objects mapped to "other" equivalence class

Points-to Analysis

$$\frac{[e := \text{equivclass } k]_i \quad \forall A \in pts(k).\ \Gamma \vdash A : \text{String}}{\text{StringEquivAbs} \in pts(e)}\ [\text{PTEquivString}]$$

$$\frac{[e := \text{equivclass } k]_i \quad \forall A \in pts(k).\ \Gamma \vdash A : \text{Integer}}{\text{IntEquivAbs} \in pts(e)}\ [\text{PTEquivInt}]$$

$$\frac{\begin{array}{l}[e := \text{equivclass } k]_i \\ \forall A \in pts(k).\ ((\Gamma \vdash A : \tau) \wedge (\text{Resolve}(\tau, .\text{equals}()) = \text{Object.equals}())) \\ K \in pts(k)\end{array}}{K \in pts(e)}\ [\text{PTEquivIdentity}]$$

$$\frac{\begin{array}{l}[e := \text{equivclass } k]_i \\ A \in pts(k) \quad \Gamma \vdash A : \tau \\ (\tau \neq \text{Integer}) \wedge (\tau \neq \text{String}) \wedge \\ \quad (\text{Resolve}(\tau, .\text{equals}()) \neq \text{Object.equals}())\end{array}}{\text{OtherEquivAbs} \in pts(e)}\ [\text{PTEquivOther}]$$

Figure 3.5: The `equivclass` (equivalence class) IR operator maps user (language)-level equality, such as the Java `.equals()` method, to heap abstractions during static analysis so that IR maps indexed with object identity can be used.

Equivalence Class Objects

In order to provide equality testing according to common higher-level language semantics (such as the `.equals()` method in Java or `operator==` in C++), we provide the `equivclass` operator. This operator takes a value and, conceptually, returns an object that represents its *equivalence class*, as defined by its *language-level equality*. The operator guarantees that if it returns different equivalence classes for two objects, then those objects must not compare equal according to their equality semantics. (However, it does not guarantee the converse: false-positive aliasing is acceptable in our analysis, and not every custom `.equals()` implementation can be analyzed precisely.)

In practice, these equivalence classes are well-defined for a few object types whose equality definitions we have explicitly understood and modeled, and reduce to a default fallback equivalence class for every other object type. Our system, which analyzes Java and thus obeys Java's equality semantics w.r.t. `.equals()` methods, currently models equivalence explicitly for integer and string objects, and for objects whose classes have no `.equals()` overrides.

The semantics of `equivclass`, and several inference rules that return appropriate abstractions, are shown in Fig. 3.5. The inference rules in this figure rely on the program's types: the predicate $\Gamma \vdash A : \text{String}$ means that the typing context Γ (assumed to have been provided by the compiler frontend) specifies that abstraction A has type String. We also use a function $\text{Resolve}(\tau, m)$ that resolves which concrete method implementation is the target of a given invoked method signature m on a given type τ. (This could be the implementation provided by that type, or by a parent class if not overridden.) For example, if a type τ does not override the `.equals()` method, then Resolve(τ, `.equals()`) = `Object.equals()`.

Note that when using integers or strings as map indices, only a single abstraction is returned: any string could be equal to any other string, no matter its allocation site, and likewise for `Integer` objects (boxed integers). Nevertheless, some useful additional precision will still be possible in these cases when we introduce distinctness (Chapter 4).

3.3.2 Modeling Sets: A Special Case of Maps

Given support for key-value maps, support for *sets*, or order-independent collections of unique objects, is straightforward: a set is just a map where set elements become map keys, and values are unused. The semantic model(s) for higher-level set container data types can therefore use the existing map abstraction at the IR level.

3.3.3 Lists: First-Class Sequences

Next, we include first-class *list* values. A list is an ordered sequence of object references. This definition covers implementations with $O(1)$ indexed element access, such as fixed or dynamically-sized arrays (e.g., Java's `ArrayList` or C++'s `std::vector`) and those with $O(n)$ indexed element access, such as linked lists: the API-level behavior is the same. An IR-level list object always contains a sequence of object references (pointers), which can in turn refer to any type of object.

Supported operations are shown in the left half of Fig. 3.6. The operations can be broadly classified into two categories: *indexed* operations, i.e. those that operate on a particular index in the list, and *non-indexed* operations, i.e. those that operate on some variable location depending on list state, such as a list append.

Analysis via Decomposition to Map Operations

We can begin our analysis of programs that use list operations by noting that a list is simply a map indexed by sequential integers. Thus, operations that are indexed, such as `listget` and `listput`, directly correspond to the analogous map operations. In fact, this is exactly how we extend our analysis to handle lists as well as maps. A list allocation is directly translated to a map allocation, and indexed list operations are translated as shown in Fig. 3.6.

In order for this decomposition to work properly, we need to extend the points-to analysis to create heap abstractions for *integer indices* as well as ordinary heap objects. This is straightforward: we simply create a single heap abstraction representing all integers and enter it into the points-to set of every integer-typed IR variable. By itself, this will result in a fairly imprecise analysis, because a map that models a list will have

IR operation	Pseudocode	Equivalent Map Operations (for Analysis)
l := listnew	*l.map := {}* *l.length = 0*	*l := mapnew*
listclear l	*l.map := {}* *l.length = 0*	*mapclear l*
listput l, idx, v	*l.map[idx] := v*	*mapput l, idx, v*
v := listget l, idx	*v := l.map[idx]*	*v := mapget l, idx*
listinsertidx l, idx, v	*l.map[idx+1..] =* *l.map[idx..]* *l.map[idx] = v* *l.length++*	*idx := virtualindex l* *mapput l, idx, v*
listremoveidx l, idx, v	*l.map =* *l.map[..idx] +* *l.map[idx1..]* *l.length--*	*(none: analysis is conservative & does not track elements by explicit indices, so index shift & removal do not affect analysis results)*
listappend l, v	*l.map[l.length] = v* *l.length++*	*idx := virtualindex l* *mapput l, idx, v*
listprepend l, v	*l.map[1..] =* *l.map[0..]* *l.map[0] = v* *l.length++*	*idx := virtualindex l* *mapput l, idx, v*
v := listpopfront l	*v = l.map[0]* *l.map = l.map[1..]* *l.length--*	*idx := virtualindex l* *v := mapget l, idx*
v := listpopback l	*v = l.map[l.length-1]* *l.map = l.map[..l.length-1]* *l.length--*	*idx := virtualindex l* *v := mapget l, idx*

Key

Pseudocode	
list[i..j]	sublist from index i (incl.) to index j (excl.)
l1 + l2	list concatenation

Figure 3.6: IR operators provided to manipulate first-class lists. Lists are analyzed as maps, and all list operations are lowered to map operations, with the help of virtual indices (§3.3.3) for non-indexed operations.

only one heap abstraction as key, namely this single integer abstraction. However, as we will see in the next chapter, additional annotations can describe the *distinctness* of individual instances of this abstraction, and so enable distinctness of objects in lists to be inferred.

The main difficulty in translating list operations to map operations arises with *non-indexed* operations, namely list append, prepend, pop-front and pop-back. All four of these operations fundamentally operate on an index that depends on the list state: append and pop-back operations depend on list length, and prepend and pop-front write to every index because they shift list contents to the right or left.

We could simply model non-indexed operations as an access to *every* index in the list. However, this results in undesired imprecision: in particular, when the analysis encounters a loop that appends elements to a list once per iteration, the analysis should somehow be able to conclude that each iteration writes to a different index in the list.

In order to properly model this behavior, we introduce the concept of a *virtual index*. The virtual-index operator takes a list instance as an argument and returns an integer. This integer is guaranteed to be unique for each invocation of the operator on a *particular* list instance as long as the only element accesses are to the latest index returned by this operator. However, if these conditions are not met, the virtual index is allowed to reset to zero, or some other reused value. (Exactly which value is not specified.) These semantics are designed such that an *append* operation can be written as a list write to the index returned by the virtual-index operator. As we noted above, we do not expect to compile the IR with data-structure operations directly to executable code, so we do not need a more precise definition that deterministically fixes the value returned. We will see in the next chapter how this definition is sufficient to analyze the behavior of a loop of append operations, as described above, with the desired precision.

3.3.4 Iterators and Traversals

Finally, our IR includes *iterators*, which represent the current position in a sequence traversal. Iterators traverse over map keys and set elements when returned from `mapkeyiter` and list elements when returned from `listiter`. The supported operations are shown in Fig. 3.7, along with the corresponding rules for points-to analysis. The program can use the `iterhasnext` and `iternext` operators to loop over all values in a sequence.

The analysis explicitly recognizes such loops over the values returned by an iterator and marks them for additional analysis: these loops become candidates for parallelization at a later stage of program analysis. One can think of such a loop over iterator values as another built-in operator, though it is not written explicitly as such.

At analysis time, an iterator is simply represented as a list, and the elements of the iterated-over data structure are copied from the map points-to set of the source list to the map points-to set of the iterator, indexed by the integer abstraction. An `iternext` operator is then just a pop-front operator.

3.3.5 Semantic Dependencies and Commutativity

We now turn to Problem 2 identified in §3.1.2 above: an analyis that is not aware of data-structure operations usually does not have enough information to determine the *semantic* dependencies and possibilities for reordering allowed by the API-level specifications of data structure implementations.

One may define a *dependency order* over dynamic occurrences of IR statements that is implied by their execution semantics such that a reordering of their execution that respects this partial order will result in the same program output. The simplest dependency order is the one that has an edge between any two

IR operation	Pseudocode	Points-to Analysis
x := iterhasnext i	*x := maplength i ≠ 0*	*(none)*
x := iternext i	*x := listpopfront i*	*(none)*
x := iterremove i	*l := maplength i* *x := listremoveidx i, l*	*(none)*
i := mapkeyiter m	*i := keys(m)*	$\dfrac{[\text{i := mapkeyiter m}] \quad M \in pts(m) \quad pts(M.K) \neq \emptyset}{A \in pts(i) \quad K \in pts(A.Ints)}$ [PTKeyIter]
i := listiter l	*i := copy(l)*	$\dfrac{[\text{i := listiter l}] \quad L \in pts(l) \quad V \in pts(L.Ints)}{A \in pts(i) \quad V \in pts(A.Ints)}$ [PTMapGet]

Figure 3.7: IR operators provided to manipulate first-class iterators. Iterator operators are lowered to list operators for analysis, and elements are simply copied to the iterator (list) points-to sets from the appropriate sources.

operations, at least one of which is a write, to the same heap location. Respecting this dependency order while parallelizing a loop is equivalent to respecting the no-heap-overlap condition described in the prior chapter that is sufficient for sound parallelization.

We wish to account for data structures with *commutative* operations, however. By modeling commutativity explicitly in the dependency order, we can address Problem 2 directly. Our system does exactly this, in two stages: first, it models *logical heap locations* such as map value slots explicitly, so that the commutativity of logically-unrelated operations to different keys is exposed; and second, it enables semantic models to define *virtual values*, which are opaque abstractions of data-structure state and can have explicitly commutative state updates.

Modeling a Map: Specific Keys and Set-of-all-Keys

The first step to exposing semantic commutativity of map operations is to name memory locations in maps semantically rather than by low-level address. This is possible because first-class map objects abstract away implementation-level details that may cause conflicts. We call this map read indexed by a logical key a *logically-indexed memory access*. Fig. 3.8 illustrates the side-effects of normal memory accesses to object fields and logically-indexed memory accesses to maps, and the dependency partial ordering between them.

The above definitions model dependencies on fields and on particular map value slots, which cover the majority of necessary ordering dependencies. However, we are not yet done. Consider the case where a program performs a number of writes to a map that insert new key-value pairs. It then iterates over the keys in the map, performing some computation on those keys (not necessarily reading the values at those keys). This map iteration is logically dependent on, i.e., cannot be reordered with, each individual key-value pair

IR operation	Modeled Side-Effect
x := y.f	Read on V.f
x.f := y	Write on V.f
mapput m, k, v	Write on V[K] CommWrite on V.keys
v := mapget m, k	Read on V[K]
i := mapkeyiter m	Read on V.keys
(other map ops analogously)	
v := newvirtval	
vvread v	Read on V.<virtval>
vvwrite v	Write on V.<virtval>
vvcommwrite v	CommWrite on V.<virtval>

Side-Effect Types

Read on V
- Ordered after prior Writes and Commutative Writes
- Commutes with Reads

Write on V
- Ordered after prior Reads, Writes and Commutative Writes

Commutative Write on V
- Ordered after prior Reads and Writes
- Commutes with Commutative Writes

Commutes?	*R*	*W*	*CW*
R	X		
W			
CW			X

Figure 3.8: The IR operations defined in this chapter have an implicit partial ordering describing the dependencies between heap accesses. The left half of this figure shows the side-effects of each operation, "natural" and synthesized by *virtual values* employed by models. The right half shows how these operations are ordered, defined by the relative commutativity of different operation types. Note that the points-to analysis itself does not use the dependency edges; they become relevant later, e.g. when analyzing loop parallelizability.

insertion, because each of the insertions adds to the iterated-over keys. However, the insertions themselves are logically independent of each other, as long as the written-to keys do not overlap. Our solution above will not capture this dependency because, if the iteration itself never reads the value at any key, it will never access the logical memory location determined by the map and key abstraction. We could encode the iteration as a read of every indexed location, but this is imprecise: it does not make the distinction between dependence on the presence of a key and dependence on the value at that key.

Fundamentally, this behavior can be captured by describing the *set of all keys* as an element of data-structure state that is independent of the value indexed by each key. A key-value insertion then performs a write to this state, and an iteration performs a read. Furthermore, the writes to this state are *commutative*: they can be reordered with respect to each other, but a read cannot be reordered with any of the writes. We now describe how we model this concept in the analysis.

Virtual Values for Additional Semantics

In order to capture this map-key commutativity, and to enable semantic models to encode arbitrary dependencies in data structures that they model, we introduce the concept of a *virtual value*. This value is a black-box abstraction that describes some aspect of data structure state. The model can issue three fundamental operations on such values: reads, writes, and commutative writes. The side-effect analysis on maps automatically creates one virtual value, the *keys* field, for every map to model the commutative dependency on the "set of all keys" described above. This analysis inserts the appropriate read and commutative-write operations to this virtual value alongside any map intrinsic that depends on, or updates, the presence of any key, respectively. We also provide these operators as IR intrinsics, alongside other first-class data types, for

Map Semantic Model (excerpts)

```
model java.util.HashMap {
  // builtin @map type
  @map values;
  ctor() {
    this.values = mapnew;
  }
  Object get(Object k) {
    key = equivclass k;
    return mapget values, key;
  }
  Object put(Object k, Object v) {
    key = equivclass k;
    oldval = mapget values, key;
    mapput values, key, v;
    return oldval;
  }
}
```

Figure 3.9: An excerpt of the semantic model for `java.util.HashMap`, showing the use of IR-level maps.

the use of semantic models that require them to model other state. The operations and their commutativity with respect to the others are shown in Fig. 3.8.

3.4 Mapping Libraries to Intrinsics: Semantic Models

So far, we have described a number of data-structure operators that grant additional information to analyses when used by a program. However, existing programs were not written with these operators in mind, and will instead continue to use data structures implemented in the standard library or by the programmer. We thus need to *translate* some of the operations in the original program into first-class data-structure operators.

To enable this, we define a *semantic model* DSL. The user or library author writes semantic models that substitute for specific named classes, such as Java's `HashMap`, and provides method implementations that operate on built-in IR types with the appropriate primitives.

These semantic models are compiled to the same IR as the rest of the program, and merged with it. The model compiler also includes metadata that binds the models to their target classes. As the analysis resolves the callgraph, it takes this metadata into account, preferring a model's method implementation over the original standard library implementation where one exists.

A sample excerpt of the model for the `java.util.HashMap` class is shown in Fig. 3.9. The syntax used here resembles Java.[1] The operators such as `mapput` and `mapget` correspond directly to the underlying IR operations. The semantic model may be arbitrarily complex, and thus can model many other data structures besides lists, maps, and sets.

Note that in its current form, our system performs this translation only in one direction, from the original program to one that uses data-structure operators in place of concrete data-structure implementations. This representation is never converted back into one with concrete implementations. The high-level view is sufficient for static analysis, and analysis results that are computed on this view will hold for the original program as well as long as the translation to data-structure operators is correct. However, recompilation from this IR is interesting future work, and is likely a useful foundation for a system that chooses optimal

[1]Due to a simplistic parser, our implementation's actual semantic-model language is somewhat lower-level than what is shown here, though no less powerful.

data structure implementations based on some analysis.

3.5 Evaluation: Points-to Precision

In order to evaluate the effect of semantic models on program analysis, we analyze programs with an existing whole-program points-to analysis, both with and without semantic models. We demonstrate increased precision in many cases, and increased analysis scalability as well.

3.5.1 Methodology

We make use of the Doop [25] static-analysis framework for Java. We chose to evaluate Java programs because the heap model is simpler than lower-level languages such as C/C++, and the binary representation, JVM bytecode, retains more type information, easing analysis. However, nothing in our approach is fundamentally dependent on Java: similar first-class data types, semantic models, and an extended points-to analysis could be built for any language with pointers or object references.

Static Analysis: The Doop analysis framework includes a whole-program Andersen points-to analysis. This analysis creates a call-graph and calculates points-to sets as we described in Chapter 2, and has configurable context-sensitivity. We run a points-to analysis with 1-object-sensitivity [73], which works well in practice with object-oriented programs. In addition, because for some benchmarks, the context-sensitive analysis without semantic models cannot complete (due to memory exhaustion), we perform context-insensitive variants of the same analyses for all benchmarks.

Doop is written declaratively as a series of inference rules in the Datalog language. We extend the inference-rule set with rules of our own to calculate map points-to sets and integrate semantic model method overrides into the call-graph resolution logic.

Semantic Models: Semantic models are combined with the analyzed program to form one unified collection of program IR as analysis input. We have developed our own simple compiler `modelc` that generates Doop-compatible IR from our semantic models, along with some metadata that ties semantic model methods to the modeled class and method names.

We include semantic models for the Java standard library classes `HashMap` and `TreeMap`, `WeakHashMap`, `ConcurrentHashMap`, `ConcurrentSkipListMap`, `ArrayList`, `LinkedList`, `HashSet`, `TreeSet`, `Vector`, `Hashtable` and `Enumeration`, and the boxed integer types. More models could easily be added, but these were sufficient for our studies.

Benchmarks: We evaluate semantic-model-enhanced analysis on 29 benchmarks from several suites: DaCapo [23], JOlden [11] (a Java translation of Olden [27]), a Java implementation of portions of the Problem-Based Benchmark Suite [100], as well our *cpu* suite, a set of CPU-intensive programs individually chosen

	With 1-Object-Sensitive Contexts					*Context-Insensitive*				
		Baseline		With Models			Baseline		With Models	
Benchmark	# Sets	Elts.	/ Set	Elts.	/ Set	# Sets	Elts.	/ Set	Elts.	/ Set
dacapo.batik	91021	3244737	35.6	1311486	14.4	36825	1093113	29.7	838786	22.8
dacapo.luindex	14127	48027	3.4	34591	2.4	11230	96183	8.6	99165	8.8
dacapo.pmd	88367	—	—	1357041	15.4	20240	314334	15.5	341091	16.9
dacapo.xalan	20054	—	—	90773	4.5	1858	21035	11.3	21883	11.8
cpu.csim	40545	—	—	147969	3.6	8180	205568	25.1	167733	20.5
cpu.djbdd	15057	110074	7.3	60809	4.0	6096	128144	21.0	120602	19.8
cpu.jacc	27420	—	—	102973	3.8	6019	34109	5.7	31369	5.2
cpu.jgrapht	2750	67875	24.7	11623	4.2	2163	123596	57.1	44786	20.7
cpu.jltxmath	1778	6224	3.5	6938	3.9	1263	17022	13.5	14337	11.4
cpu.jscheme	21147	—	—	96882	4.6	2390	35824	15.0	25718	10.8
cpu.jtidy	18143	94011	5.2	79552	4.4	13061	86592	6.6	110947	8.5
cpu.sblbdd	9820	56544	5.8	62230	6.3	4156	32708	7.9	30650	7.4
cpu.sat4j	6881	16129	2.3	15628	2.3	5157	31154	6.0	41842	8.1
olden.bh	2756	7561	2.7	8320	3.0	1793	18156	10.1	15424	8.6
olden.bisort	19527	—	—	88861	4.6	1387	17101	12.3	14418	10.4
olden.em3d	19738	—	—	89168	4.5	1514	17357	11.5	14673	9.7
olden.health	2159	6819	3.2	7533	3.5	1523	17307	11.4	14624	9.6
olden.mst	2013	6480	3.2	7195	3.6	1498	17272	11.5	14588	9.7
olden.perim	2246	7191	3.2	7905	3.5	1537	17689	11.5	15004	9.8
olden.power	2137	6550	3.1	7266	3.4	1537	17348	11.3	14663	9.5
olden.treeadd	19492	—	—	88800	4.6	1329	17110	12.9	14425	10.9
olden.tsp	1995	6525	3.3	7240	3.6	1480	17317	11.7	14633	9.9
olden.voronoi	3246	16596	5.1	17258	5.3	1917	19929	10.4	16591	8.7
pbbs.compsort	1792	6192	3.5	6907	3.9	1277	16986	13.3	14302	11.2
pbbs.convhull	2038	6484	3.2	7158	3.5	1450	35471	24.5	28056	19.3
pbbs.intsort	1801	6255	3.5	6969	3.9	1281	16989	13.3	14305	11.2
pbbs.nn	19828	—	—	89215	4.5	1535	25489	16.6	26089	17.0
pbbs.raycast	2925	8394	2.9	8531	2.9	2147	33506	15.6	26593	12.4
pbbs.remdup	19678	—	—	88977	4.5	1381	23846	17.3	18330	13.3
Average (no timeouts)	9720.3	196245.7	6.6	88165.2	4.3	5125.8	96656.9	15.5	78415.7	12.0

For fair comparison, averages do not include benchmarks that timed out in any of the four runs.

Table 3.1: Relative sizes of points-to sets for a whole-program points-to analysis, with and without semantic models. Results are provided as total number of elements, or heap abstractions, in all points-to sets (left column of each pair), and the average number of elements per set (right column of each pair). Analyses with missing data in the baseline column did not complete due to memory exhaustion.

for complex control flow and data structures (in particular, compilers, parsers, and simulators). The *cpu* suite contains: circuit [44], CloudSim [26], DJBDD [68], Jacc [54], Janino [3], JGraphT [5] JLaTeXMath [6], JScheme [13], and raytracer [74].

3.5.2 Results

In order to compare points-to analyses, we use *points-to set size* as an objective measure that is a reasonable proxy for analysis precision. Intuitively, if the analysis produces smaller points-to sets while remaining sound, it is likely more precise, as long as it is not shrinking the points-to set sizes by simply aggregating objects into fewer abstractions. Because both the baseline and model-enhanced analyses use the same context sensitivity, this is not the case: the same context-qualified heap abstractions will exist in both cases. Thus, smaller points-to sets truly indicate better precision.

To compare points-to set sizes, we select variables that are present in the main program only, rather than standard-library classes, and then we compute the average number of heap-abstraction elements per set, where each (method context, variable) tuple has its own set.

Table 3.1 provides this data for both context-sensitive and -insensitive analysis variants, with semantic models in use and without. Note first of all that the original context-sensitive analysis without semantic models did not complete in many cases. We found that this was due to memory exhaustion. In general, analysis without semantic models is much more expensive, because the analysis has to work to understand library implementations of the data-structure classes. Common data structures such as sets, maps and lists tend to be used widely, so their analysis is either replicated, if context sensitivity is effective at separating each use, or objects from different use-cases blend together, if not.

The second conclusion that we draw is that on average, benchmarks analyzed with a semantic-models-enhanced points-to analysis had more precise results: an average of 4.3 elements per points-to set (context-sensitive variant), compared with 6.6 elements per set in the baseline without models. This effect is particularly significant in benchmarks that make heavier use of standard-library data structures, such as those in the *dacapo* and *cpu* suites. For example, in *batik*, an SVG (Scalable Vector Graphics) rendering library and the benchmark with the most points-to set elements overall, the average points-to set size is more than halved. These improvements both directly enhance alias-analysis precision and feed into other later analyses. For example, the analyses that we build in the following chapters can do less work because the points-to analysis produces a smaller points-to graph.

We note that in some benchmarks, the points-to sets grew slightly when semantic models were used. This is an implementation artifact: when IR-level data structures and models are enabled, the analysis maps built-in array operations to list operations as well. (This generally increases precision.) The points-to analysis for list operations must ensure that a heap abstraction exists for the key, namely, the abstraction representing all integers. We examined all cases where points-to set sizes increased in Table 3.1 and found that the integer abstractions entirely account for the increase.

3.6 Discussion: First-Class Primitives vs. DSLs

We have introduced an extension to the compiler that enables it to natively understand certain higher-level primitives, and a means to map library code to these primitives. This extension could be seen as an embedded DSL for data-structure operations, and the analyses that we perform in later chapters are able to reason about programs' data-structure-related operations just as DSL compilers can reason about the primitives in domain-specific programs. If future work extends the set of first-class data-structure types and primitives available, this data-structure-aware system converges further toward high-level DSLs.

There is an apparent conflict between this view of our contributions and our motivation for avoiding the use of DSLs in Chapter 1. However, our approach differs from that of a data-structure DSL in the use of *semantic models* to effectively map the explicit, lower-level standard library implementations to higher-level first-class constructs while permitting the code that uses the library to remain in a general-purpose language. In other words, our work can be seen in two parts: first, a translation from a general-purpose program to one that contains code in an embedded data-structure DSL, and second, a compiler that can analyze and optimize this DSL.

Based on this view, we suggest furthermore that there is significant room to optimize the system in several dimensions. First, between the extremes of fully-automatic semantic-model-based mapping and fully-manual use of hints or new DSL primitives, there are many other approaches to analyzing a programmer's intent. For example, a developer tool could implement an interactive loop with the programmer whereby it uses semantic models and other forms of pattern-matching to map as much of the program as possible to higher-level constructs, and then show a view of this mapping (potentially enriched with other analysis results) to the programmer while highlighting program operations that were not mapped successfully. The programmer could then add minimal annotations as necessary to gradually raise the level of the system's understanding until it is able to transform the program as desired.

Next, while we have proposed to analyze general-purpose code, the analyses in the following chapters of this thesis could just as well operate on, and benefit from, a program written in a DSL of some form. Our essential insight is that we must simply understand the program at the level of its data structure operations before we can analyze those operations and perform transforms; the way in which we obtain that program understanding is flexible. In general, the mapping portion of this work could consist of some combination of semantic models (pattern-matching on general-purpose code) and explicit DSL compilation stages that all map onto a shared, general set of primitives in the enhanced IR that we analyze.

3.7 Related Work

We have presented a compiler IR extension and analysis that enables direct specification of, and reasoning about, programs that manipulate common data structures such as maps, sets, and lists. A number of related approaches exist that add data-structure awareness to the compiler. However, our work is the first to explicitly provide first-class abstractions, with built-in operators for container data types in an otherwise general-purpose language and to analyze these intrinsics appropriately in heap-related analyses.

3.7.1 Explicit Semantics with DSLs

Many domain-specific languages (DSLs) have been proposed and developed to enable programmers to express high-level specifications of algorithms in various domains. The common aspect of all of these systems is that they provide high-level primitives (such as graph traversals or per-node operators, matrix multiplications, etc.) but impose semantic restrictions on their use (e.g., a graph node operator can only access neighbor nodes). In exchange, the backend can freely choose among implementations.

Among the domains with DSLs are graphs (GraphLab [69]), meshes (Liszt [39], Ebb [22]), matrices and numerical computations (Halide [86], TensorFlow [9], SPIRAL [85]), machine-learning models (OptiML [105], ScalOps [111]), compiler IR [48], distributed data sets (PigLatin [76], Scope [29], Sawzall [80]), database queries (SQL, GraphQL [2]) and general algorithms over maps, lists and sets (Galois [63]). In each of these systems, the user writes their program either in a completely custom language (e.g., SPIRAL's mathematical notation for FFT-like transforms), or using building blocks provided as a library or framework embedded in an existing general-purpose language (Halide in C++, Galois in C++ or Java, etc.).

In many cases, a DSL is built around a central data structure (such as an in-memory graph), and a key design aspect is that of *inversion of control*: the DSL runtime or library manages an entire high-level operation, such as a graph traversal, and the user only provides small, configurable pieces of the computation, such as a computation to run at every graph node. These functions or code blocks must stay within certain bounds in order to operate correctly: for example, they must access only certain subsets of the traversed data structure. (All major graph and mesh computation DSLs have restrictions in this spirit, possibly with mechanisms to permit controlled and specially managed exceptions.) In exchange, this bounded behavior provides the runtime with certain freedoms: for example, it can reorder operations more freely, because fewer dependency edges exist between individual computations.

In other cases, a DSL is built as a set of operations on data structures but top-level control remains with the user. However, a sequence of operations may be visible at once to the DSL compiler or runtime, allowing for optimization scopes across individual operations. For example, a SQL query planner typically sees an entire top-level `SELECT` statement at once, and is able to rearrange scan, filter and join steps in the overall query plan. Similarly, Halide views an entire computation pipeline (producing a matrix of pixels, for example) at once, and is able to fuse loops and eliminate intermediate storage when it is profitable to do so.

The PetaBricks system, introduced by Ansel et al. [14], is explicitly built around the central DSL idea to provide implementation choice to a backend. In particular, it provides a general framework in which a library author can describe several possible implementations of a data structure operation (such as a matrix multiplication), including tiling and decomposition strategies, enabling the compiler to make performance-based choices.

The Delite framework [28] introduces a meta-framework to construct DSLs based on a small number of fundamental primitives: a dependency graph of operators to enable scheduling, and support for collections (such as vectors and matrices) portable across different backends.

As we described above, all of these systems provide powerful optimization opportunities to an execution backend. However, fundamentally, this power comes at the cost of explicit user effort to specify higher-level semantics, and to conform to operator usage restrictions that may limit the range of applications that can be expressed.

3.7.2 Data-Structure or API Semantics in the Compiler

A number of works introduce data structures, and in particular *collection* data structures (such as maps, lists, and sets), as compiler intrinsics in order to provide the compiler a deeper view into program behavior. In contrast to our system, the most common application of this idea in prior work is to enable optimization of data-structure choice: i.e., if the compiler or runtime system knows the requirements (e.g., types of traversals, queries, and updates), the access patterns, and perhaps the distribution or other characteristics of the average data-structure contents, then it can make informed choices about the implementation both at the macro-scale (which data structure to use) and micro-scale (how to tune parameters or build indices).

Several systems approach this problem by inserting the new functionality behind the API boundary: e.g., CollectionSwitch [35] includes a runtime that implements common Java data structure classes and profiles usage of data structures. Chameleon [99] works similarly, and can then suggest implementation options to the programmer interactively. This work can also be done offline: the Brainy system [57] collects a log (trace) of activity and then analyzes it offline to make an informed decision. Smart Data Structures [40] collects statistics and makes decisions based on a machine-learning model. Darwinian Data Structures [20] instead pits competing data structures against each other in a performance competition, using a genetic algorithm to evolve the best choices.

Another line of work requests more explicit information from the programmer, making the best choices using this explicit semantic guidance. This can take the form of a DSL-like framework: for example, Just-in-Time Data Structures [37] gives the programmer facilities to describe multiple forms of a data structure and conversion functions between them, along with heuristics that guide runtime shifts between these representations. This process can also begin with high-level descriptions of data-structure needs or behavior. For example, Idreos et al. [51] introduce a novel description of common data structures from first principles, with simple building blocks. Given this high-level description and reasoning at the level of the basic building blocks, the system can compute costs of various options and choose the best ones.

Both of the above lines of work share our insight (and that of DSLs) that a high-level description of the required data-structure behavior, i.e., at the API level, provides implementation freedom leading to higher performance. However, as noted above, the goal fundamentally differs. This prior work and ours are compatible, and potentially symbiotic: if the programmer has used high-level data structure operators, and semantic models map many parts of the program's heap to these data structures, then the problem of data-structure choice and replacement becomes much easier. Conversely, choosing an appropriate data structure based on actual requirements (updates and queries performed) may remove artificial constraints that were hindering e.g. loop parallelization from transforming the program as desired.

A number of works enable the programmer to describe the semantics of an API directly (whether for a data structure or for some other interface, such as a set of system calls). Kulkarni et al. [63, 62] introduce a means of annotating *commutativity* on APIs of data structures in the Galois framework, enabling the system to reason about operation reorderings while executing loop iterations in a parallel framework. This commutativity model permits the system to avoid aborts of optimistic concurrency when a reordering is safe. Clements et al. [33] annotate commutativity of the system call interface, reasoning about operation reorderings as a means to evaluate multicore scalability (an interface whose semantics grant commutativity of more operations leads to a more scalable system). Both of these works share our insight (and that of many past works, such as Rinard and Diniz [92] and Aleen and Clark [10]) that recognizing commutativity is important for parallelization or otherwise finding scalability.

Wu and Padua [115] introduce data structure models in the context of loop parallelization. Their proposal is mostly static, but requires insertion of some runtime checks to determine whether (e.g.) hash-table keys might alias. Our analysis as presented in this chapter differs from theirs in that it provides greater flexibility to model arbitrary data structures with semantic models, and provides a number of useful primitives beyond hash-tables (maps) provided by this prior work. We will also see in later chapters that our system can statically prove parallelizability in a systematic way (using distinctness) where Wu and Padua's system required dynamic checks.

3.7.3 Mapping Programs to DSLs

Some work has been done to express general-purpose programs that solve problems in a domain into primitives in an appropriate domain-specific language (DSL): e.g., Kamil et al. [58] automatically extract regular numerical kernels from Fortran programs into the Halide DSL [86, 87]. This system operates by pattern-matching: it identifies particular data structures in memory as well as strings of instructions that produce the contents of those data structures. For a limited domain, such as loop-based numeric computations with regular structure, this can work well, though it is difficult to extend to more irregular programs. Our approach differs in that it relies on some programmer, usually the standard library or other library author, to provide explicit models. This shifts work from the analysis author (who builds pattern recognizers) to the library author, but provides much greater flexibility, and reduces complexity by avoiding the need to reverse-engineer behavior from low-level implementation details.

3.7.4 Program Analysis with Data-Structure Awareness

One aspect of semantic models that appears in past work to some degree is the use of data-structure awareness to enhance analysis precision. Though no work explicitly provides built-in maps and lists with first-class points-to analysis understanding as we do, others either automatically derive data-structure invariants and high-level shape descriptions, or else enable the user to do so. Shape analysis [55, 46, 112, 21] attempts to find descriptions of heap-object connectivity, such as linked-list or tree patterns, to enhance other analyses' precision. This differs from our work in that it expends effort to derive what we ask the user or library author to tell us directly with semantic models. In addition, the shape descriptions carry no meaning or other semantic implication other than the invariants that comprise them. In other words, one cannot directly understand that an operation on a tree is a binary-tree insertion and understand its commutativity from shape analysis alone.

Other systems enable the user to specify data-structure invariants with a logic of some sort, such as the Jahob [64] system. This approach is similar to that of semantic models in that it has first-class notions of data structures. The main difference is that these data structures exist for modeling invariants only, and so may not completely describe the behavior of the program. In contrast, if a program uses the first-class primitives provided by our system, then an analysis built on our system can understand the program's dataflow from end to end.

3.8 Chapter Summary

In this chapter, we introduced *first-class data structure values*, enabling the compiler to reason about maps, lists, iterators, and several other objects on which programs operate with built-in operators. We have shown how our system enables the user to map standard library classes to these operators with *semantic models* in an extensible way. Finally, we demonstrated via empirical evaluation that these techniques significantly enhance the precision of a points-to analysis framework on Java programs.

Chapter 4

Daedalus: Enhanced Alias Analysis with Distinctness

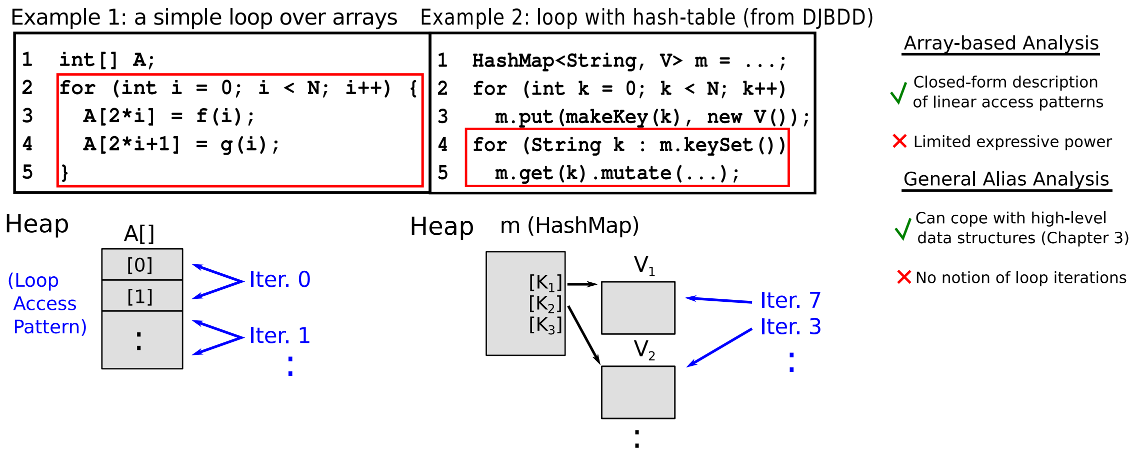
A pointer once pointed, distinctly
To objects summarized succinctly
As bits of the heap
Quickly built and quite cheap
And rapidly deref'd quite unlinked-ly.

So far, we have described an enhanced points-to analysis that achieves its improved precision by analyzing data structure operations directly as first-class compiler intrinsics rather than by analyzing the implementations of these operations. While we have shown this system to be effective at its stated goal, we have not yet applied it to the loop-parallelization problem. We now do so by refining the points-to results with respect to loops using an intuitive concept that we call *distinctness*.

4.1 Alias Analysis for Loop Parallelization

Recall from §2.3 that non-aliasing across iterations of a loop is sufficient for that loop to be parallelizable. Fundamentally, a loop parallelization analysis is a specialized alias analysis, focusing (beyond ordinary analysis such as that of Andersen [12]) on aliasing between loop iterations. We must first answer the question: why are standard alias analyses insufficient to resolve this aliasing?

Most loop parallelization-focused analyses today can be seen as an alias analysis that reasons about arrays and indexing of those arrays, as we covered in §2.1. These analyses often cannot describe the behavior of programs that use other data structures because *the abstraction is insufficient*: simply, there is no closed-form description of *which* object is accessed in a given loop iteration, so the access cannot be modeled by the analysis.


Example 1: a simple loop over arrays
Example 2: loop with hash-table (from DJBDD)
Array-based Analysis
✓ Closed-form description of linear access patterns
✗ Limited expressive power
General Alias Analysis
✓ Can cope with high-level data structures (Chapter 3)
✗ No notion of loop iterations
Heap
A[]
[0]
[1]
(Loop Access Pattern)
Iter. 0
Iter. 1
Heap
m (HashMap)
[K1]
[K2]
[K3]
V1
V2
Iter. 7
Iter. 3

```
int[] A;
for (int i = 0; i < N; i++) {
  A[2*i] = f(i);
  A[2*i+1] = g(i);
}
```

```
HashMap<String, V> m = ...;
for (int k = 0; k < N; k++)
  m.put(makeKey(k), new V());
for (String k : m.keySet())
  m.get(k).mutate(...);
```

Figure 4.1: Comparison of array-based reasoning and alias-based reasoning on two example loops. Array-based reasoning is able to prove parallelizability for loops with regular indexing patterns, as in Example 1. However, it encounters two issues on Example 2: non-array data structures, and inability to express *which* object is mutated by each iteration. Alias analysis as in Chapter 3 can address the former but not the latter.

4.1.1 Array-Based Reasoning Has Limited Scope

To see a concrete example of this, consider the two examples in Fig. 4.1. We illustrate the heap objects used by each program and the target loop's access pattern on each. Note that *both* loops are parallelizable, because each iteration has a non-overlapping set of accessed memory locations. In particular, in the second example, the loop accesses each allocated `V` instance exactly once, and never visits any instance twice.

The first program is a straightforward loop nest with linear array accesses. Traditional array-based reasoning can succinctly represent this access pattern and prove that each array write does not alias any other array access in other loop iterations. However, it encounters two issues when analyzing the second example. First, it cannot understand that the key-value map data structure is an indexed access, so even if the key were a linear function of an index variable, it would not be represented and analyzed as such. Our contribution in Chapter 3 addresses this problem by adding first-class key-value map support. In principle, a linear array access analysis could be adapted to analyze map accesses too.

However, another problem, less easily solved, also occurs: the array-based analysis has no way of representing the access pattern of the second example's loop. The invariant (the loop visits each object once) does not lead to a closed-form expression describing *which* object is visited.

Our observation is that such invariants are often important. In particular, this invariant in the target loop arises from an invariant in the *map*: the first loop in Example 2 inserts a set of distinct `V` objects into the map, so the map has unique values at each key by construction. Programs often maintain such aliasing invariants that are relevant to parallelization. In addition, the second loop in Example 2 never visits a key more than once. This property arises through the combination of the `.keySet()` method (which returns a

```
1 for (i=0; i<N; i++) {
2   Data d1 = new Data(); // A1
3   Data d2 = new Data(); // A2
4   d1.f = f1();
5   d2.f = f2();
6 }
```

Points-to Sets: pts(d1) = { A_1 }
pts(d2) = { A_2 }

Figure 4.2: Attempting to compute loop parallelizability via aliasing or points-to analysis: a conventional points-to analysis provides enough information to conclude that the store at line 4 does not alias the store at line 5 (and vice-versa), but not that it doesn't alias itself across iterations.

set, guaranteeing no duplicates) and the traversal loop over its returned value. Combining these facts lets us conclude that `m.get(k)` in the second loop returns a different value every iteration.

4.1.2 Aliasing Analysis Is Loop-Agnostic

One could apply a more general alias analysis, such as one based on points-to analysis, to the problem instead: for example, an analysis that had an understanding of data structures might make more progress analyzing a program that uses a key-value map. As we address *loop parallelization*, we are particularly interested in aliasing across iterations of the same loop. Let us consider how we might analyze the loop in Fig. 4.2. The points-to sets produced by Andersen points-to analysis are shown.

The relevant aliasing question is: do either of the stores (at lines 4 and 5) alias with any stores or loads in a different iteration? The answer is intuitively no, because new `Data` instances are allocated each iteration, and are not carried across iterations. Using a conventional points-to analysis, however, we cannot arrive at this answer. The analysis can disambiguate `d1` and `d2`, proving that one never aliases the other, because they have disjoint points-to sets. However, the analysis is not flow- or loop-sensitive, so any non-empty points-to set for a variable describes its value in *all* iterations. Thus, according to the analysis, any store could alias itself across iterations.

At least one past work addresses this problem by adding some notion of indexing to the heap abstraction, analogous to the conventional array-access analysis [114]. However, this quickly becomes untenable as program complexity grows: in many programs, there is no closed-form expression for *which* object is visited each iteration. However, we *can* often prove that each iteration visits a *different* object than all the others. This is the fundamental insight of DAEDALUS.

4.1.3 Our Approach: Distinctness, a Form of Non-Aliasing

To address this shortcoming, we extend a general alias analysis (with data-structure awareness, as described in the prior chapter) in the direction of the array-based aliasing analysis for parallelization (i.e., to answer cross-iteration aliasing questions). Our *key insight* is that the array-based analysis tries to *prove too much.* A closed-form description of all accesses is unnecessary: we only need to know that iterations access *different* memory locations.

We thus define *distinctness analysis*, a principled extension of may-point-to analysis that annotates the points-to graph with loop-relative information. Each such annotation states, essentially, "this reference will be to a different object each iteration of loop L." Such an annotation on a pointer is exactly what is necessary in order to parallelize a loop that writes through this pointer, because it ensures that the pointer will not alias itself in other iterations. (It may alias other pointers in other iterations; we address this with a must-alias analysis, as described in §4.4.)

Returning to the second example program in Fig. 4.1, we see that in the first loop, we can immediately establish that the value inserted into the map is distinct every iteration, because it is freshly allocated. Next, we can observe that the key-value map is mutated only by this first loop, and that at each iteration, it inserts a unique value. Thus the system can derive the invariant that the values in the map have no duplicates, because a duplicate would require the insertion of the same value more than once. Finally, in the second loop, we can observe that the key `k` is distinct each iteration. We can combine the map invariant and this latest distinctness fact to derive a new distinctness fact regarding the value `m.get(k)`.

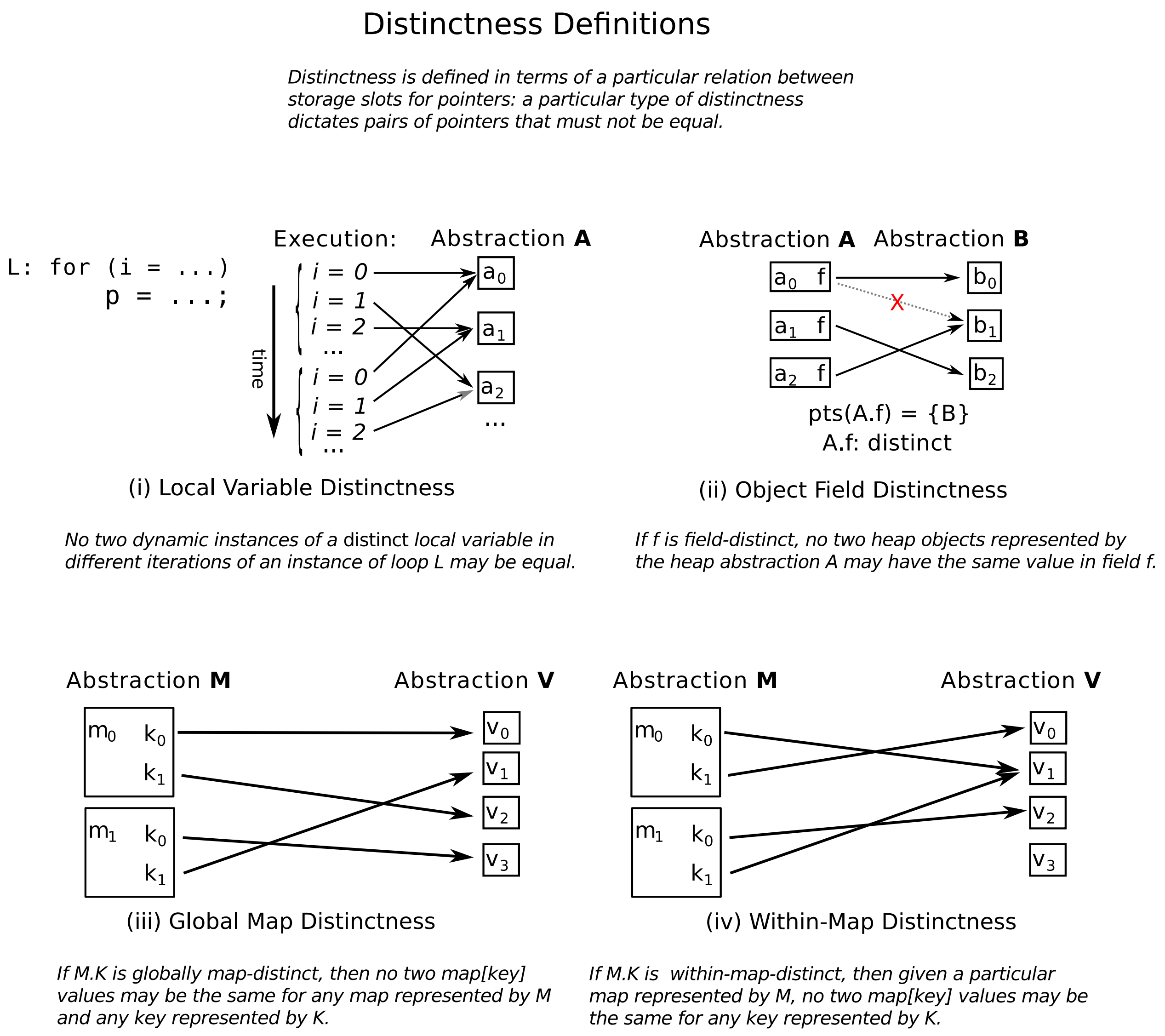
4.2 Distinctness Analysis: Definitions and Analysis Rules

In this section, we formally define *distinctness analysis* and provide a set of simplified analysis rules. A formal description of the entire analysis and a soundness proof can be found in an appendix to this dissertation (Appendix A).

Distinctness analysis, which is the main contribution of this chapter, is a specific type of alias analysis that computes, in a low-cost way, the necessary information for program parallelization. It takes as input the results of a points-to analysis, and augments program variables and heap abstraction fields with analysis results that further refine the results of the points-to analysis.

4.2.1 Distinctness: Non-Aliasing Within a Scope

Our analysis derives several types of *distinctness*: distinctness on *local variables*, on *object fields* for a given heap abstraction, and two types of distinctness on *map values* for a given pair of map and key heap abstractions. Fig. 4.3 illustrates these four types of distinctness, which we now define formally.



Distinctness is defined in terms of a particular relation between storage slots for pointers: a particular type of distinctness dictates pairs of pointers that must not be equal.

No two dynamic instances of a distinct *local variable in different iterations of an instance of loop L may be equal.*

If f is field-distinct, no two heap objects represented by the heap abstraction A may have the same value in field f.

If M.K is globally map-distinct, then no two map[key] values may be the same for any map represented by M and any key represented by K.

If M.K is within-map-distinct, then given a particular map represented by M, no two map[key] values may be the same for any key represented by K.

Figure 4.3: Definitions of four types of distinctness: local variable, object field, global-map, and within-map distinctness.

Variable Distinctness

First, in order to capture this notion of "different object every iteration," we define *variable distinctness*. We say that a variable v is *distinct with respect to a loop L* if, within a given instance of L, v never takes on the same value in two different iterations.

A variable distinctness judgment, $\text{Distinct}(v, L)$, is defined more precisely as follows:

Definition 1. *Consider any two iterations i, i' of* one execution *of loop L. Given variable v, take any pointer values v_i, $v_{i'}$ from iterations i and i'. (A particular variable may have multiple values in one iteration if defined in a nested loop.)*

$\boldsymbol{Distinct}(v, L)$	Must not alias self across iterations in same instance:	$i \neq i' \;\rightarrow\; v_i \neq v_{i'}$

We also define a related property, *constantness*, derived alongside distinctness. Denoted $\text{Constant}(v, L)$, this is more specific than constantness in a traditional constant-propagation pass: it does not indicate that v is *statically constant*, but rather *constant over one instance of loop L*. This is defined precisely as follows:

Definition 2. *Consider any two iterations i, i' of* one execution *of loop L. Given variable v, take any pointer values v_i, $v_{i'}$ from iterations i and i'.*

$\boldsymbol{Constant}(v, L)$	Must alias self across iterations in same instance:	$i \neq i' \;\rightarrow\; v_i = v_{i'}$

Because distinctness and constantness facts will meet at control-flow merges with an *intersection* meet function, for simplicity in the implementation we actually compute the *negation* of these judgments. This allows our rules to avoid $\forall$-quantifications, and works correctly in the presence of cycles in the dataflow graph.

Definition 3. *Given the method M containing v, and the relation $LoopInMethod(M, L)$ that holds for every loop in the body of M:*

$\boldsymbol{NotDistinct}(v, L)$	$LoopInMethod(M, L) \wedge \neg Distinct(v, L)$
$\boldsymbol{NotConstant}(v, L)$	$LoopInMethod(M, L) \wedge \neg Constant(v, L)$

Using Variable Distinctness for Parallelization

Distinctness is directly motivated by the loop parallelization problem: given a distinctness fact stating that v takes on a different value in iterations of L, we can safely parallelize a loop with a store to v. In general, for a loop to be parallelizable, all pointers to which the loop body performs a store must be distinct with respect to the loop. (We will return to the parallelization problem at the end of this chapter (§4.5) after developing the inference rules for distinctness itself.)

Note, however, that distinctness alone is not sufficient to parallelize loops with *more than one* pointer variable that points to a given abstraction. In such a case, even if each pointer variable is individually distinct with respect to the loop, one pointer in one iteration may alias the *other* pointer in another iteration (distinctness makes no claims about aliasing between different variables). We return to this problem in §4.4 and resolve it with careful application of a must-alias analysis.

Field Distinctness

So far, we have considered only variables. However, in most programs, pointers are stored on the heap as well. A pointer may refer to an object with a tree of sub-objects; if we know that particular heap invariants hold, we should be able to transfer a distinctness fact on that root pointer to distinctness facts on derived pointers to sub-objects accessed via fields.

Field distinctness differs from variable distinctness in one fundamental way. Variable distinctness is relative to a loop L, because variables are defined at a program point within the loop. However, data structures can outlive any particular loop: they are frequently built by one loop and traversed by another, often much later. We thus need a domain aside from the iterations of a single loop instance over which we can reason about distinctness. We choose the simplest option: all objects represented by a heap abstraction.

A heap abstraction's field is *field-distinct* if, for every represented object, the field has a different value (see Fig. 4.3). This is useful because it propagates distinctness through a load: if a pointer p is distinct w.r.t. loop L, then the result of field load $p.f$ (given a distinct field) is distinct, too.

We define field distinctness as follows:

Definition 4. *Let X be a may-alias heap abstraction. We say that $x \in X$ if object instance x is represented by X.*

FieldDistinct$(X.f)$	$\forall x, x' \in X \quad x \neq x' \rightarrow x.f \neq x'.f$
	No two objects in abstraction can have aliasing field f

Similarly to variable distinctness, we actually compute the negative judgment FieldNotDistinct$(X.f)$, but we will use both forms below for simplicity.

Map Distinctness

Finally, we consider key-value maps, and by extension, lists (implemented by reducing their operations to map operations as described in §3.3.3) and sets (implemented by storing set elements as map keys as described in §3.3.2). Supporting distinctness for values stored in maps, lists, and sets is very important to analysis precision because programs often use these heap data structures to maintain collections of distinct objects. By inferring that the objects stored into a container data type are distinct, and then using this when the program iterates over the container to produce variable distinctness facts, we can accurately analyze such programs.

One can consider a map value slot to be, fundamentally, a *two-dimensional memory location identifier*: both the particular map object, and the key used to index that map, identify the value to be accessed. We can thus define, for each (map, key) tuple of heap abstractions, several types of value distinctness, using different definitions of the domain over which the values must be distinct. First, *global map distinctness* indicates that the value is different across all concrete (map, key) tuples of a map object and a key object represented by the map and key heap abstractions. In contrast, *within-map distinctness* indicates that the value is different for every key in a single map (but the same value may appear in another map in this abstraction). The latter is more limited, but is often possible to derive where global map distinctness is not, and it can still produce many distinctness facts in practice. Fig. 4.3 illustrates both.

We define the two types of map distinctness as follows:

Definition 5. *Let M be a may-alias heap abstraction for a map, and K be a heap abstraction for a key in that map. We say that $x \in X$ if object instance x is represented by X.*

MapGlobalDistinct$(M.K)$ $\forall m, m' \in M \quad \forall k, k' \in K \quad (m, k) \neq (m', k') \rightarrow m[k] \neq m'[k']$

No two value slots in any map $m \in M$ indexed by any key $k \in K$ can have aliasing values.

MapWithinDistinct$(M.K)$ $\forall m \in M \quad \forall k, k' \in K \quad k \neq k' \rightarrow m[k] \neq m[k']$

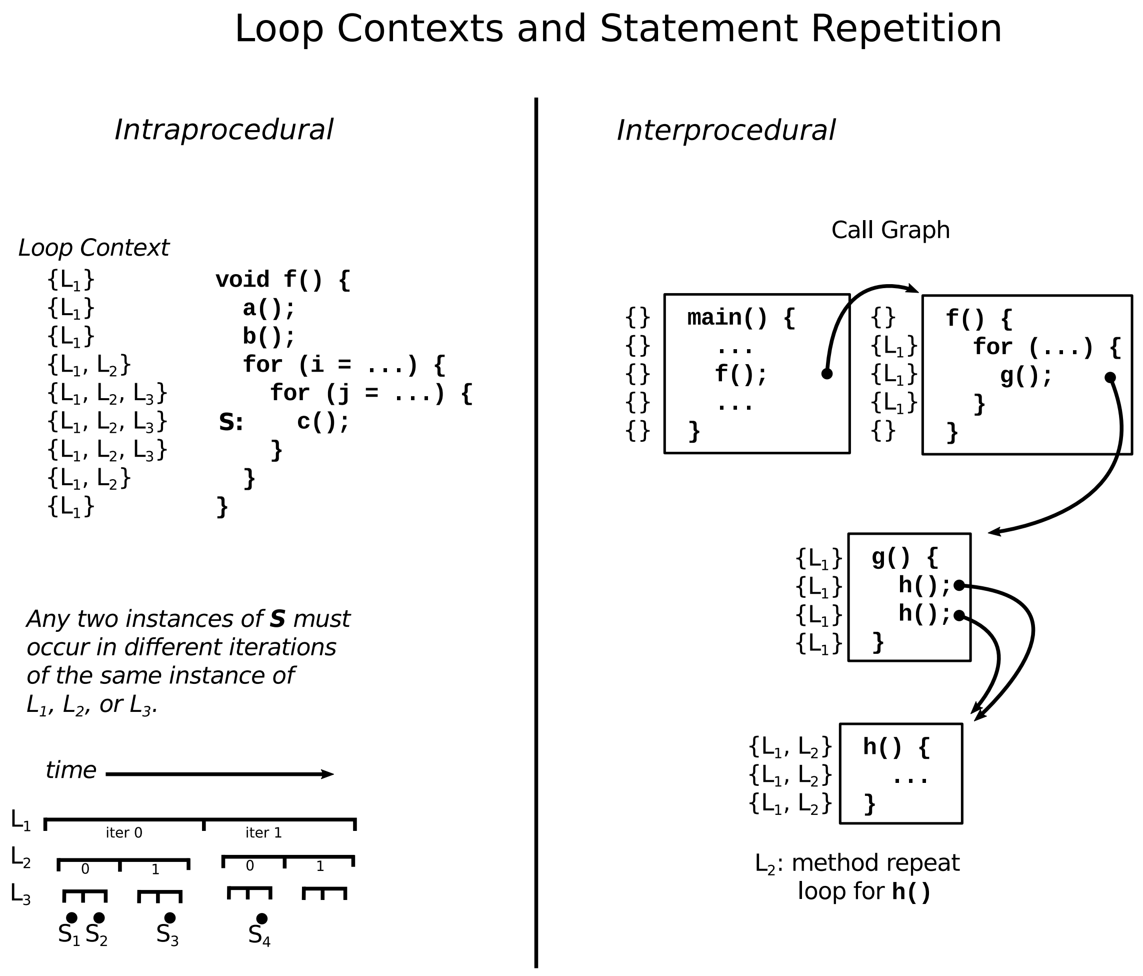
No two value slots in a particular map $m \in M$ indexed by any key $k \in K$ can have aliasing values.

For completeness, we note that a third type of distinctness, symmetric to within-map distinctness, also exists: *within-key distinctness* describes the situation where the value is different for every map (corresponding to an abstraction) indexed by a given key. However, this type of distinctness is less frequently seen, and for simplicity in analysis implementation, we do not attempt to derive or use it.

4.2.2 Loop Nests: Reasoning About Repetition

Before we can derive any distinctness facts, we need to reason about *repetition* of program statements during execution, because repeating a statement's side-effects (e.g., appending to a list) may result in a duplicate pointer, hence non-distinctness, in a heap data structure. The reader may refer to Fig. 4.4 as an illustration of the definitions below.

Fundamentally, we reason about repetition in terms of *loops*. Every statement S has a *loop context* $\mathbb{L}(S)$ which is a set of loops. The loop context is initialized with all *natural loops* in the statement's method


Loop Contexts and Statement Repetition
Intraprocedural
Interprocedural
Loop Context
Call Graph
Any two instances of S must occur in different iterations of the same instance of L1, L2, or L3.
time
L1
L2
L3
iter 0
iter 1
0
1
0
1
S1 S2
S3
S4
L2: method repeat loop for h()

```
{L1}                void f() {
{L1}                  a();
{L1}                  b();
{L1, L2}              for (i = ...) {
{L1, L2, L3}            for (j = ...) {
{L1, L2, L3}        S:    c();
{L1, L2, L3}            }
{L1, L2}              }
{L1}                }
```

```
{}  main() {         {}    f() {
{}    ...            {L1}    for (...) {
{}    f();           {L1}      g();
{}    ...            {L1}    }
{}  }                {}    }

{L1}  g() {
{L1}    h();
{L1}    h();
{L1}  }

{L1, L2}  h() {
{L1, L2}    ...
{L1, L2}  }
```

Figure 4.4: Loop-context definitions. A statement's loop context *captures all repetition*: that is, any two dynamic instances of a statement during some execution must be in different iterations of the same instance of some loop in the statement's loop context.

that contain that statement, and also includes any loops that are in the *method loop context*, i.e., define the repetition of the method itself. We say that the loop context *captures all repetition*: any particular dynamic instance of a statement must be in a different iteration of the same instance of at least one of the loops in its context. We thus define:

Definition 6. *We define* $\mathbb{L}(S)$ *to be the* loop context *of statement* S*. A loop* L *that contains* S *is in its loop context:* $S \in LoopBody(L) \Rightarrow L \in \mathbb{L}(S)$*. Furthermore, all loops in the* method loop context $\mathbb{L}(M)$ *of the method* M *containing* S *are in the context:* $\mathbb{L}(M) \subseteq \mathbb{L}(S)$*.*

We initialize the *method loop contexts* to maintain the invariant as follows: if a method has only one callsite in the callgraph, its method loop context inherits all loops from the caller's call statement loop context. Otherwise, if it has more than one callsite in the callgraph, it inherits all loops in the *intersection* of the loop contexts of all callers' call statements, as well as a new synthetic *method repeat loop*. The method repeat loop does not actually exist in the program. Rather, it is an analysis fiction that virtually has one instance, and one iteration within that instance for every time the method is called. It is necessary because with more than one callsite, even the loops in all callers' loop contexts do not capture all possible repetition. (Recursive and co-recursive methods will also have a method repeat loop in context because a co-recursive cycle must have at least one method with more than one caller, or else the cycle is unreachable.)

Appendix A contains a proof of the above "captures all repetition" invariant.

4.2.3 Inferring Variable Distinctness

We now describe several of the inference rules that compute distinctness for program variables. We first develop some analysis rules to prove distinctness in cases of object allocation and variable assignment. We will assume for now that the program is a single procedure (i.e., we will give an intraprocedural analysis), and extend this to the interprocedural case in §4.2.6. We assume the program is in an imperative SSA (static single assignment) IR (intermediate representation), with heap objects that have named fields.

Loop Nest: First, we can immediately introduce some *Constant* and *NotDistinct* facts for any variable relative to loops that are below the variable definition point in the loop nest. We define InnerLoop(S_i) to be the innermost natural loop surrounding statement S_i, and we write $L' \subset L$ to indicate that L' is a subloop of L: that is, the body of L' is completely contained within the body of L.

Rule 1. *All variables are not distinct w.r.t. any subloop of their definition point.*

$$[\textsc{SubLoopNotDistinct}]\ \frac{[x := \ldots]_i \qquad L = \mathit{InnerLoop}(S_i) \qquad L' \subset L}{\mathit{NotDistinct}(x, L')}$$

Rule 2. *All variables are not constant w.r.t. any loop that is not a subloop of their definition point, unless produced by a statement type explicitly handled elsewhere.*

$$[\textsc{SubLoopNotConstant}]\ \frac{\begin{array}{ll} [x := \ldots]_i & S_i \in M \\ \neg S_i = [x := y] & L = \mathit{InnerLoop}(S_i) \\ \neg S_i = [x := \phi(\ldots)] & \mathit{LoopInMethod}(M, L') \\ \neg S_i = [x := y.f] & L' \not\subseteq L \end{array}}{\mathit{NotConstant}(x, L')}$$

Object Allocation: The first inference rule, [Alloc] in Rule 3 below, instantiates a distinctness judgment for the result of an object allocation (**new** operator) for every loop in the statement's loop context. This follows because the allocation returns a new object, different from all others. We write the rule first in terms of a positive-polarity judgment Distinct(x, L):

Rule 3. *A newly allocated object is distinct per iteration in all containing loops.*

$$[\textsc{Alloc}]\ \frac{[x := \mathit{new}\ T]_i \qquad L \in \mathbb{L}(S_i)}{\mathit{Distinct}(x, L)}$$

but it actually exists as an exception in a fallback rule that creates NotDistinct facts for all other variables and loops in the method:

Rule 4. *Any variable not produced by some other statement type handled with an explicit rule is not distinct w.r.t. all loops in the method.*

$$[\textsc{Fallback}]\ \frac{[x := \ldots]_i \qquad \begin{array}{l} S_i \neq [x := \mathit{new}\ T] \\ S_i \neq [x := \phi(\ldots)] \\ S_i \neq [x := y.f] \\ S_i \neq [x := \mathit{mapget}\ m, k] \\ S_i \neq [x := \mathit{mapput}\ m, k, v] \\ S_i \neq [x := \mathit{equivclass}\ y] \\ S_i \neq [x := \mathit{virtualindex}\ l] \end{array} \qquad \begin{array}{l} S_i \in M \\ \mathit{LoopInMethod}(L, M) \end{array}}{\mathit{NotDistinct}(x, L)}$$

Variable Assignment: Because we analyze SSA IR, all single- and multiple-input assignment is captured by ϕ-nodes. We thus write a rule in Rule 5 that *merges* distinctness at ϕ-nodes. In fact, expressed in the negative (*NotDistinct*) sense, assignment is very simple: if any input to the ϕ-node is not distinct with respect to L, then the result is not, either. Note that we only propagate not-distinct judgments for loops that remain in context at the ϕ-node.

Rule 5. *If any assignment source is not distinct w.r.t. L, then the result is not, either.*

$$[\textsc{AssignDistinct}]\ \frac{[x := \phi(x_1, \ldots, x_n)]_i \qquad \mathit{NotDistinct}(x_i, L) \qquad L \in \mathbb{L}(S_i)}{\mathit{NotDistinct}(x, L)}$$

Loop Induction Variables: We also introduce distinctness on loop induction variables over an arithmetic sequence. This is implemented as a simple pattern recognition (a loop-carried recurrence on a variable through a single add of an integer constant) that then injects the appropriate distinctness fact.

4.2.4 Variable Constantness

We derive variable constantness facts for program variables in a simple way based on program structure. The [SubLoopNotConstant] rule in Rule 2 above produces a NotConstant(v, L) judgment for a variable relative to every loop that is not a subloop of L, unless that variable's constantness is handled explicitly by another rule, i.e., for assignments and loads. In other words, an SSA variable defined outside the scope of a loop remains constant across iterations of that loop, but not relative to other loops. Then, not-constantness propagates across assignments in the same way that not-distinctness does:

Rule 6. *If any assignment source is not constant w.r.t. L, then the result is not, either.*

$$[\textsc{AssignConstant}]\ \frac{[x := \phi(x_1, \ldots, x_n)]_i \qquad \mathit{NotConstant}(x_i, L) \qquad L \in \mathbb{L}(S_i)}{\mathit{NotConstant}(x, L)}$$

4.2.5 Field Distinctness

Proving Field Distinctness at Stores

In order to prove field distinctness, we must examine all stores that could create a pointer-pointee edge by assigning to f, and reason about the pointer and value variables provided to the store. If, when a store writes to the same abstraction and field, either the stored value is distinct for each dynamic store instance, or a possibly non-distinct value overwrites the same location, then the field is distinct.

The rules [StoreOverlap] (Rule 7) and [Store] (Rule 8) implement this. First, [StoreOverlap] finds any abstraction and field written by more than one store. We disqualify these fields because it is difficult to reason about aliasing between values from different stores. (For example, two different store statements may individually create a distinct pointer-pointee relationship by storing unique pointer values into a list of fields, but intermixing these two pointer value sequences results in repetition and thus non-distinctness.)

Next, the [Store] rule encodes the intuition above: if for any loop in context of a store (i.e., that may result in a repetition of the statement), the pointer has not been proven constant (so could differ between instances) and the pointee has not been proven distinct (so could alias between instances), then two different object instances' fields could point to the same pointee. This would result in a non-distinct field. Conversely, if either the pointer is constant or the pointee is distinct over all store instances, then the field is distinct.

Rule 7. *More than one store to the same field on the same abstraction results in field non-distinctness: we cannot reason about how two sequences of stores will overlap.*

$$[\textsc{StoreOverlap}]\ \frac{\begin{array}{ll} [x.f := y]_i & \\ [x'.f := y']_{i'} & X \in \mathit{pts}(x) \\ i \neq i' & X \in \mathit{pts}(x') \end{array}}{\mathit{FieldNotDistinct}(X.f)}$$

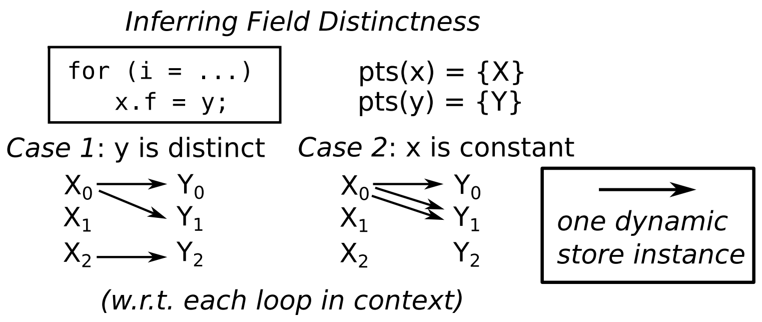


Figure 4.5: Inferring field distinctness from stores.

Rule 8. *For all stores to a particular abstraction and field, if no store overlaps with another store to that abstraction and field, and if for each store, for each loop in that store's context, either the pointee is distinct or the pointer is constant, then the field is distinct. Thus, if w.r.t. some loop in context, the pointer is not constant* and *the pointee is not distinct, then the field is not distinct.*

$$[\textsc{Store}]\ \frac{[x.f := y]_i \qquad L \in \mathbb{L}(S_i) \qquad \begin{array}{c} \mathit{NotConstant}(x, L) \\ \mathit{NotDistinct}(y, L) \end{array} \qquad X \in \mathit{pts}(x)}{\mathit{FieldNotDistinct}(X.f)}$$

Together with the fact that by definition, the loop context captures all possible repetition of a statement, this condition ensures distinctness of the field f. We illustrate this with the help of Fig. 4.5. Consider how non-distinctness could arise: two different pointer objects (x variable in figure) must refer to the same pointee object (y variable). This would require y to vary (be non-constant) while x repeats a value (is non-distinct). Seen another way, the field will be distinct if we either write a new value every time we perform a store (Case 1 in the figure), or if, when we may repeat a stored value (non-distinct y), we *ensure* that we overwrite the old value via the constant x pointer (Case 2).

Using Field Distinctness at Loads

We can now use field distinctness at loads: if a field is distinct on all heap abstractions in the pointer's points-to set, and the pointer itself is distinct w.r.t. a given loop, then the loaded value becomes distinct as well. The [LoadBasePtr] and [LoadField] rules (Rules 9 and 10 below) formalize this. In addition, we require the [LoadConflict] rule (Rule 11) to handle the case where two different heap abstractions' fields can point to the same object: in this case, even if each field were individually found to be distinct, the same pointee object might be reachable from two different objects, one represented by each heap abstraction.

Rule 9. *If a pointer is not distinct in L, a value loaded through that pointer is not distinct in L either.*

$$[\textsc{LoadBasePtr}]\ \frac{[x := y.f]_i \qquad \mathit{NotDistinct}(y, L)}{\mathit{NotDistinct}(x, L)}$$

Rule 10. *If a field is not distinct, any value loaded from that field is not distinct w.r.t. any loop in context, regardless of the base pointer.*

$$[\textsc{LoadField}]\ \frac{[x := y.f]_i \qquad Y \in \mathit{pts}(y) \qquad \mathit{FieldNotDistinct}(Y.f) \qquad L \in \mathbb{L}(S_i)}{\mathit{NotDistinct}(x, L)}$$

Rule 11. *If two different heap abstractions can point to the same pointee, we cannot conclude distinctness even if each field is individually distinct.*

$$[\textsc{LoadConflict}]\ \frac{\begin{array}{lll} & Y_1 \in \mathit{pts}(y) & \\ & Y_2 \in \mathit{pts}(y) & \mathit{pts}(Y_1.f) \cap \mathit{pts}(Y_2.f) \neq \emptyset \\ [x := y.f]_i & Y_1 \neq Y_2 & L \in \mathbb{L}(S_i) \end{array}}{\mathit{NotDistinct}(x, L)}$$

4.2.6 Interprocedural Support

As noted above, we have simplified some aspects of the Daedalus system so far for clarity. We now describe several details related to the context-sensitive interprocedural aspect of the analysis.

First, Daedalus supports context sensitivity (for our configuration, 1-object sensitivity [73] as implemented in Doop [25]), with Daedalus's contexts matching those of the points-to analysis. To do so, we (i) parameterize every variable distinctness fact on method context, and (ii) parameterize every heap object reference with heap context. This follows the standard approach described in §2.5.2.

Next, Daedalus supports interprocedural analysis. This builds on the callgraph provided by the points-to analysis, as described in §2.5.1, and relies upon the interprocedurally-derived loop contexts described in §4.2.2. Once loop contexts are properly established for all methods, the only remaining glue to tie together each method's analysis is to generate synthetic assignment statements that connect call arguments to formal method parameters, and method return values to callsite result variables. When multiple callsites can call a single (context-qualified) callee, the multiple inputs are merged with ϕ-nodes on each parameter. Likewise, when a single callsite can call multiple callees, the return value is merged from all callees with a ϕ-node.

4.3 Distinctness in Maps, Sets and Lists

4.3.1 Inferring and Using Map Distinctness

We derive map distinctness in an analogous way to stores. Because there is an additional "dimension" to the stored location (a map and a key, rather than solely a pointer), the required combination of distinctness and constantness facts is slightly more complex.

Recall that there are two forms of map distinctness: *global map distinctness* and *within-map distinctness*. First, consider global map distinctness. This property implies that a given value in a map is not repeated in *any* other map, at *any* other key, for all maps and keys represented by the map abstraction and key abstraction in question. To derive such a fact, analogously to the store inference rule, we have:

Rule 12. *More than one map store to the same map and key abstraction results in global and within-map non-distinctness.*

$$[\textsc{MapStoreOverlap}]\ \frac{\begin{array}{ccc}[\mathit{mapput}\ x_1, y_1, z_1]_i & \multirow{2}{*}{$i \neq j$} & X \in (\mathit{pts}(x_1) \cap \mathit{pts}(x_2)) \\ [\mathit{mapput}\ x_2, y_2, z_2]_j & & Y \in (\mathit{pts}(y_1) \cap \mathit{pts}(y_2))\end{array}}{\mathit{MapNotDistinct}(X[Y]) \qquad \mathit{MapNotDistinctWithinMap}(X.Y)}$$

Rule 13. *For all maps and keys represented by a given map and key abstraction, a map is globally distinct if the sole* **mapput** *to operate on that (map, key) tuple either stores a distinct value, or stores to a constant map* and *key, for every loop in context. Thus, if for some loop in context, the* **mapput** *stores a non-distinct value and either the map pointer or key is not distinct, then the map's contents are not globally distinct.*

$$[\textsc{MapStoreNotDistinct}]\ \frac{[\mathit{mapput}\ x_1, y_1, z_1]_i \quad \begin{array}{c}\mathit{NotDistinct}(z_1, L) \\ \mathit{NotConstant}(x_1, L) \vee \\ \mathit{NotConstant}(y_1, L)\end{array} \quad \begin{array}{c}X \in \mathit{pts}(x_1) \\ Y \in \mathit{pts}(y_1)\end{array}}{\mathit{MapNotDistinct}(X.Y)}$$

We can derive within-map distinctness similarly. The key difference is that the analysis does not need to prove that the stored-to map is constant if the value is not distinct (so the requirements are weaker than for global distinctness: every within-map-distinct map is globally distinct, but not vice versa).

Rule 14. *For all maps and keys represented by a given map and key abstraction, a map is within-map distinct if the sole* **mapput** *to operate on that (map, key) tuple either stores a distinct value, or stores to a constant key, for every loop in context. Thus, if for some loop in context, the* **mapput** *stores a non-distinct value to a non-constant key, then the map's contents are not within-map distinct.*

$$[\textsc{MapStoreNotDistinctWithinMap}]\ \frac{\begin{array}{l}[\mathit{mapput}\ x_1, y_1, z_1]_i \\ \mathit{NotDistinct}(z_1, L) \\ \mathit{NotConstant}(y_1, L)\end{array} \quad \begin{array}{c}X \in \mathit{pts}(x_1) \\ Y \in \mathit{pts}(y_1)\end{array}}{\mathit{MapNotDistinctWithinMap}(X.Y)}$$

4.3.2 List Support: Distinctness of Virtual Indices

As we described in §3.3.3, lists are reduced to maps using *virtual indices* for non-indexed operations such as appends. This virtual index represents an arbitrary index that is unique per dynamic use, so that an append writes a new slot each time, but resets for each new concrete instance of the indexed object (e.g., list). We thus create a distinctness fact for a virtual index w.r.t. any loop in the local loop context for which the indexed list is *constant*:

Rule 15. *A virtual-index variable is distinct exactly when the indexed object (e.g., list) is constant. Thus, it is not distinct whenever the indexed object is not constant.*

$$[\textsc{VirtualIndexDistinct}]\ \frac{[x := \mathit{virtualindex}\ l]_i \qquad \mathit{NotConstant}(l, L)}{\mathit{NotDistinct}(x, L)}$$

4.3.3 Iterations over Sets, Map Keys, and Lists

We can derive distinctness of the *iterator value* in an iterator loop by tracking distinctness of an iterator sequence itself. The iterator is reduced to a list (and thus, in turn, a map), and distinctness facts are transferred as appropriate.

An iterator over map keys (and thus also over set elements, modeled as map keys, as noted in §3.3.2) is always a distinct sequence. In contrast, an iterator over list elements has a distinct sequence (distinct-within-map on the underlying map) exactly when the source list does.

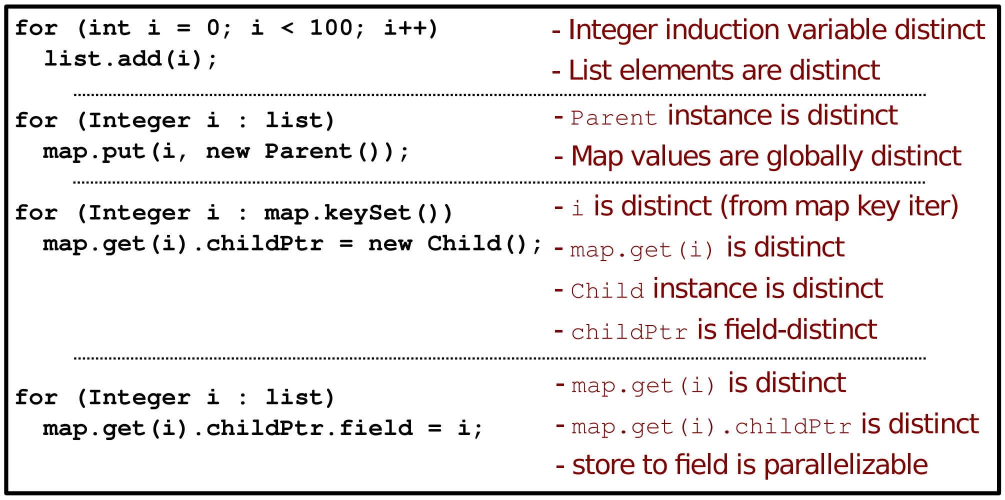
4.3.4 Equivalence Classes: Distinctness of Map Keys

Recall from §3.3.1 that the IR includes an operation to map key-value map indexing semantics, which are object identity-based (i.e., keyed on the pointer value itself), to other language semantics that define equality using other (value-based) semantics. This operation, `equivclass`, returns a virtual object, synthesized for static analysis only, that represents an *equivalence class* for a key. The IR-level map can then be indexed using this equivalence class. Because many IR-level maps will be indexed in this way, we must derive the distinctness of map keys produced by `equivclass` in order to have useful precision in real programs.

Daedalus can derive distinctness facts in three cases. First, the operator passes through distinctness facts on any *integer* abstraction to describe the result as well. Conceptually, there is a single *abstraction* for equivalence class objects for integers, and the result of the operator is a different object in this abstraction for each different integer input.

Rule 16. *If the argument to* `equivclass` *is always an integer and is distinct, then the resulting equivalence class is distinct as well. Thus, if the argument is always an integer and is not distinct, then the result is not distinct.*

$$[\textsc{EquivClassInt}]\ \frac{[x := \mathit{equivclass}\ y]_i \qquad \begin{array}{c}\forall A \in \mathit{pts}(y).(\Gamma \vdash A : \mathit{Integer}) \\ \mathit{NotDistinct}(y, L)\end{array}}{\mathit{NotDistinct}(x, L)}$$



```
for (int i = 0; i < 100; i++)          - Integer induction variable distinct
  list.add(i);                          - List elements are distinct

for (Integer i : list)                 - Parent instance is distinct
  map.put(i, new Parent());            - Map values are globally distinct

for (Integer i : map.keySet())         - i is distinct (from map key iter)
  map.get(i).childPtr = new Child();   - map.get(i) is distinct
                                        - Child instance is distinct
                                        - childPtr is field-distinct

for (Integer i : list)                 - map.get(i) is distinct
  map.get(i).childPtr.field = i;       - map.get(i).childPtr is distinct
                                        - store to field is parallelizable
```

Figure 4.6: Example of distinctness analysis.

Next, we pass through distinctness facts when the types of all objects that could be arguments to the `equivclass` operator are *identity* objects – that is, objects without overridden equality methods, so that the high-level equality semantics are also object-identity-based.

Rule 17. *If an object's type has no overridden equality semantics (so is object-identity-based by default), non-distinctness facts on the object pass through to its equivalence-class object.*

$$[\textsc{EquivClassIdentity}]\ \frac{\begin{array}{ll} & \forall A \in \mathit{pts}(y). \\ [x := \mathit{equivclass}\ y]_i & (\Gamma \vdash A : \tau \wedge \mathit{Resolve}(A, \mathit{.equals()}) = \\ \mathit{NotDistinct}(y, L) & \qquad \mathit{Object.equals()}) \end{array}}{\mathit{NotDistinct}(x, L)}$$

Finally, all other equivalence class objects are non-distinct.

Rule 18. *All arguments to* `equivclass` *that are not captured by either of the above two rules lead to non-distinct* `equivclass` *results.*

$$[\textsc{EquivClassOther}]\ \frac{\begin{array}{ll} & A \in \mathit{pts}(y) \\ [x := \mathit{equivclass}\ y]_i & \Gamma \vdash y : \tau \\ L \in \mathbb{L}(S_i) & \tau \neq \mathit{Integer} \wedge \\ & \quad \mathit{Resolve}(A, \mathit{.equals()}) \neq \mathit{Object.equals()} \end{array}}{\mathit{NotDistinct}(x, L)}$$

4.3.5 Example: Distinctness in Data Structures

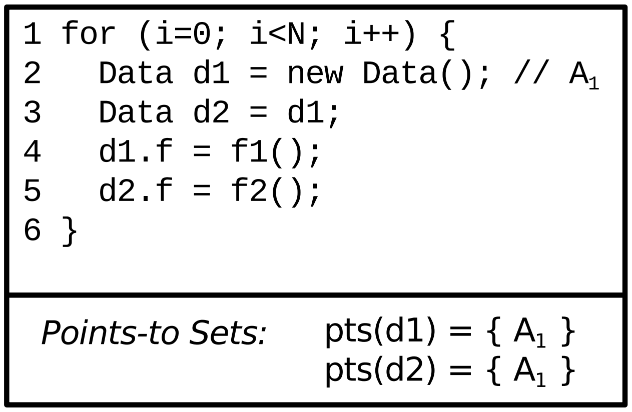
We demonstrate the power of the proposed analysis with an example. Fig. 4.6 shows a program fragment with four loops. Consider the first loop: We know that `i` is distinct w.r.t. the loop because it is an incremented induction variable. We can propagate this distinctness through the list append (which decomposes to a virtual index and a map store). We then examine the second loop, an iterator loop over the list, and use this list's key-distinctness to note that `i` is distinct. The newly allocated `Parent` is also distinct, as all allocations are. Thus, we also infer key-distinctness for `map`. Next, the third loop fetches the key-set and iterates over its elements. Map keys in the key-set are always distinct, so `i` is distinct here. As we access the key-distinct map with distinct keys, the values (`Parent` objects) are also distinct. Hence, when we store the `Child` to its field, we can infer a distinctness implication on this field. Then finally, when we iterate again over distinct `Parent` objects in the fourth loop, we know that the fetched `Child` is distinct, so the store to its field is parallelizable (each store will go to a different object).

This conclusion is beyond the reach of conventional array-based analyses because the map and list data-structures are not represented in the model. Furthermore, the goal of such an analysis – to precisely determine *which* object or array element is accessed each iteration – would be difficult to attain: the iteration over the map may visit keys in an arbitrary order. Only by deriving *distinctness* do we attain a feasible analysis, and this is sufficient for parallelization.

4.4 Which Distinct Value?: Must-Alias Analysis Inside Loops

So far, we have given an analysis that can prove that a variable does not alias itself across loop iterations. Thus, a loop whose body consists of a store through a pointer variable that is distinct (with respect to that loop) will be parallelizable. However, as we briefly described in §4.2.3, distinctness alone does not suffice when *more than one* store to a particular heap abstraction occurs within a loop body: in this case, distinctness can tell us that each store does not alias itself across iterations, but not that one store in one iteration does not alias another store in another iteration.

To see this concretely, consider the example program in Fig. 4.7. This example resembles Fig. 4.2, except that both pointers `d1` and `d2` point to the same object in each iteration. Distinctness analysis will be able to prove that each of these variables is distinct with respect to the loop at line 1: `d1` directly, because it is a new allocation, and `d2` by assignment from `d1`. Thus, each variable will not alias itself across iterations. However, this is not enough to safely parallelize the loop. In particular, `d1` in one iteration may alias `d2` in another iteration. Note that we have no invariants relating the *specific* object pointed to by `d1` to the *specific* object pointed to by `d2`, only that they refer to the same heap abstraction. Intuitively, each variable



```
for (i=0; i<N; i++) {
  Data d1 = new Data(); // A1
  Data d2 = d1;
  d1.f = f1();
  d2.f = f2();
}
```

Points-to Sets: pts(d1) = { A_1 }
pts(d2) = { A_1 }

Figure 4.7: Modified version of Fig. 4.2 demonstrating the need for must-alias analysis. Distinctness analysis alone will be able to prove that `d1` does not alias itself across iterations, and likewise for `d2`, but not that `d1` in one iteration does not alias `d2` in another iteration.

could point to the same series of distinct objects in the loop's iterations, but in permuted orders. We need to somehow identify that the two variables traverse the distinct object sequence in the *same* order so that we know this cross-iteration aliasing is not possible.

In order to address this need, we apply a simple *must-alias* analysis based on tag propagation. This analysis produces conclusions that differ from *may-alias* facts, such as that provided by Andersen points-to analysis, because they refer to specific objects. Whereas a may-alias analysis can soundly make claims as coarsely as it likes (in the limit, claiming that every variable in the program *may* alias), a must-alias analysis errs in the direction of claiming *nothing*: we don't know for sure that any two variables *must* alias. By employing such an analysis, when the analysis claims that (e.g.) `d1` and `d2` must alias, we know for sure that the above cross-iteration aliasing cannot occur.

The analysis works on a simple principle: it propagates *tags* describing particular program values along the dataflow (variable assignments). Each SSA variable has a set of one or more tags. When two different tags meet as inputs to a merge-point (e.g., a ϕ-node), a new tag is generated instead, except for one case that we describe below. It is always safe to generate a new tag: it simply discards some information. Slightly simplified versions of the inference rules for our must-analysis are given below. The rules generate results that uphold the following invariant: if two variables have the same tag, and only that tag, *and* the may-alias analysis indicates that they *may* alias (they have at least one heap abstraction in common), then they *must* hold the same value. (A full description with all details, including an additional refinement to the rules to handle field loads and a precise specification of the upheld invariant and corresponding soundness proof, can be found in Appendix A.)

Rule 19. *A newly allocated object receives a new tag.*

$$[\textsc{TagAlloc}]\ \frac{[v_i\ :=\ new\ T]_i}{Tag[v_i] = \{Alloc_i\}}$$

$$Orig[Alloc_i] = S_i$$

Rule 20. *An assignment from a single source simply propagates the tag from source to destination.*

$$[\textsc{TagAssignSingle}]\ \frac{[v_i\ :=\ v_j]}{Tag[v_i]\ =\ Tag[v_j]}$$

Rule 21. *An assignment from multiple sources, such as a $\phi-$node, that may not alias (according to a may-alias analysis) generates a new tag.*

$$[\textsc{TagAssignMultiNonOverlap}]\ \frac{\begin{array}{c}[v_i\ :=\ \phi(v_{j_1}, v_{j_2}, \ldots, v_{j_n})]_i \\ \forall k, l.pts(v_{j_k}) \cup pts(v_{j_l}) = \emptyset\end{array}}{Tag[v_i] = \{Assign_i\} \qquad Orig[Assign_i] = S_i}$$

Rule 22. *An assignment from multiple sources where at least two sources may alias propagates tags from all inputs.*

$$[\textsc{TagAssignMultiOverlap}]\ \frac{\begin{array}{c}[v_i\ :=\ \phi(v_{j_1}, v_{j_2}, \ldots, v_{j_n})]_i \\ \exists k, l.pts(v_{j_k}) \cup pts(v_{j_l}) \neq \emptyset\end{array}}{Tag[v_i] = \cup_{m=1}^{n} Tag[v_{j_m}]}$$

The rules encode the above-described behavior for allocations and single-input assignments. The one notable detail is the handling of multiple-input assignments (ϕ-nodes). There are two strategies that could be employed at these assignments to generate tag sets for the assignment results. The assignment could propagate all tags from inputs to outputs. Then either all inputs have one tag, so they must all alias, so we can propagate the tag to the output because it will alias all inputs, or else there is more than one tag among the inputs, in which case we propagate more than one tag and the tag-aliasing invariant does not apply (it applies only to variables with one tag). Or, instead of this always-propagate strategy, the analysis could always generate a new tag, which is trivially correct (it makes no claims about aliasing). We use a heuristic to choose between these options to maximize precision on common program idioms: we generate a new tag when inputs with all separate heap abstractions join (these will certainly have more than one tag among inputs, reducing precision otherwise), and propagate all tags when possibly-aliasing inputs join (this pattern occurs sometimes when pointers are passed through recursive methods, creating cyclic assignment graphs).

Now that we have *distinctness* results that indicate when a pointer must not alias itself across iterations, and *must-alias* results that indicate when two pointers must be the *same* distinct value in an iteration, we are ready to reason about cross-iteration aliasing in general to find parallelizable loops.

4.5 Parallelizing Loops Using Distinctness

We now define conditions necessary to parallelize a loop. We provide a full set of inference rules and a soundness proof in Appendix A. This section is a high-level summary of the parallelizability analysis, providing *five conditions* that must be met for a loop to be parallelizable.

4.5.1 Heap Non-Interference: Distinct Stores

The **first condition** for loop parallelization is that no loop-carried dependencies exist through the heap. Daedalus examines distinctness facts on pointer and index variables in order to determine whether this condition holds. (Distinctness is actually a more general property, but only distinctness facts on pointers and indices matter for the loop-parallelization problem.) If for every abstraction accessed by a load or store within the loop body, either (i) no store accesses this abstraction, or (ii) all loaded and stored pointers to this abstraction are distinct w.r.t. this loop, and they *must alias* according to the must-alias analysis, then no dependencies exist. This applies to both *scalar* (object field) and *indexed* (map with key) accesses. In the latter case, distinctness of either the map or the key is sufficient.

4.5.2 Aggregate Data Structure State

The **second condition** is that no dependencies exist through properties of built-in data structures (maps and lists): e.g., a map key iteration depends on *all* prior insertions, because they may insert new keys. As described in §3.3.5, we explicitly capture *data-structure-level* fields, such as "all keys in the map," and effects on them: e.g., a `mapput` writes "all keys," while a `mapkeyiter` reads it. Note, however, that two map inserts usually commute (as long as their keys are distinct). To model this, we add a new effect type used by *mapput*: a *commutative store* reorders with other commutative stores, but not with loads or regular stores. (Likewise, loads reorder with loads, but not with other operations.) Note that any stronger ordering of *mapput* operations on the same key is already handled above as a dependence between two indexed accesses.

The first and second conditions together can be seen as a *side-effect analysis*: we can summarize the effects of every IR statement as a load, store, or commutative store to either a *scalar location*, i.e., field on a particular heap abstraction, or an *indexed location*, i.e., map heap abstraction indexed by key heap abstraction. These side-effects are described in more detail in §3.3.5. The analysis can then examine all side-effects acting on each scalar or indexed location and detect parallelization-inhibiting combinations.

Note that in our implementation, this side-effect analysis also uses a conventional array-indexing-based parallelization analysis to augment results. Specifically, if the indexing analysis can prove that array accesses are parallelization-safe (w.r.t. themselves and other array accesses), then these array accesses are excluded from consideration by the distinctness-based rules. This enhancement improves precision but is not strictly necessary, as it is orthogonal to the core analysis.

4.5.3 Local Variables, Arrays, and Parallelizability

The **third condition** is that no loop-carried local variable dependencies can exist, except for *reduction* patterns [81]: a commutative operator (addition, multiplication, bitwise-AND/OR, min/max) and no use of the intermediate value.

We also analyze array indices using a simple analysis that propagates affine function descriptions of integers in terms of a loop integer induction variable: e.g., if $i = 0 \ldots n$ in a loop, we understand distinctness of array indices like $3i + 5$.

The **fourth condition** for parallelization is that the iteration space can be computed prior to the loop: this is true for integer-increment and iterator loops.

4.5.4 Global Side-Effects

Finally, the **fifth condition** is that the loop body should not have any global side-effects. We define a global side-effect as any effect not accounted for by the usual heap-access rules. In our system, this category consists only of calls that are not resolved in the callgraph, because the callee does not exist within the analysis input IR: for example, calls to native JVM code. In particular, this condition will exclude all loops that perform IO because IO requires system call(s) through native code.

Once all five conditions are satisfied, Daedalus produces a directive that describes the parallelizable loop. The compiler backend can then use this directive to extract the loop body and parallelize the loop.

4.5.5 Locking

In the above side-effect analysis, we have reasoned with respect to the program *as seen by static analysis*, with *semantic models* replacing the actual data structures for maps, lists, sets, etc. in the standard library. In practice, many of these data structures will not be thread-safe by default. To ensure that the parallelization semantics of the model and the actual implementation match, we insert *fine-grained locking* around any callsite reachable from a parallelized loop body that invokes a method covered by a semantic model. This ensures that its effect occurs atomically, as the analysis assumes of built-in IR instructions that operate on data structures. This locking insertion, together with accurate models and side-effect modeling of IR instructions, permits commutative API semantics (e.g., those of set or map inserts) to be leveraged when parallelizing loops.

4.6 Evaluation

4.6.1 Methodology

To demonstrate that Daedalus finds useful parallelism, we evaluate the analysis by measuring speedup with simulation on a set of Java benchmarks. Our implementation is built on the Doop [25] static-analysis framework, and is (like Doop) written in Datalog. There are 532 inference rules on 292 relations specific to Daedalus. We use an in-house microarchitectural simulator on instruction traces derived from a modified OpenJDK, with a loop parallelization transform on the trace, to evaluate runtime performance. We also evaluate real-system speedup of one benchmark (§4.6.4).

Performance evaluation: In this work, the Daedalus analysis, a form of pointer alias analysis, is our primary contribution. However, unlike many other alias-analysis works, because our analysis has a special emphasis on the loop parallelization problem, we choose to evaluate *end-to-end performance* on parallelized programs. To do so, we report two main statistics: *parallelizable coverage*, or the fraction of dynamic instructions that are in the body of any parallelizable loop, and *parallelized speedup*, the result of actually parallelizing the loop. Coverage is measured simply by using the results of our analysis to count instructions in a dynamic instruction trace. To measure performance, we use a *microarchitectural simulator*. This grants us increased ability to experiment (e.g., to measure the effect of iteration spawn latency), and enables us to avoid several real-world engineering challenges that are orthogonal to (and outside the scope of) this work. For example, in Java, a loop-iteration closure needs to allocate an object to hold captured state, which has overhead. This could be avoided by implementing a parallel loop primitive at the JVM bytecode level, likely the right choice for a production implementation of Daedalus's backend. However, we can model the performance without this engineering effort by assuming a configurable iteration spawn latency for the runtime.

In this simulation methodology, once we run Daedalus to identify parallelizable loops (using the analysis workflow described below), we (i) collect an instruction trace from the interpreter loop of a modified JVM, (ii) *demultiplex* the trace by chopping it into one work-chunk per loop iteration of a parallelizable loop, and (iii) schedule these work-chunks onto cores of the simulator at runtime according to their dependencies (and respecting locks). The simulator models a detailed microarchitecture, and translates JVM bytecodes to μops with renaming of JVM operand stack slots to virtual registers, producing execution similar to that of JITted machine code. Core and memory hierarchy parameters are given in Table 4.1.

Parallel Runtime: Our simulation is agnostic to the parallel runtime, simply modeling a fixed per-iteration "task spawn" latency of 10 cycles by default. We evaluate sensitivity in §4.6.3. The runtime could use low-latency distributed software workqueues with job-stealing, as in Cilk [24], hardware-assisted scheduling, as

Cores	1 – 16 cores, out-of-order, respects dependencies, 256-entry ROB, 4-wide dispatch/retire, 4 int/3 FP ALUs
L1	64 KB, 4-way, 64B blocks, MESI, 3-cycle latency
LLC	8 MB, 8-way, 15-cycle latency
DRAM	256 banks, 100/200-cycle (8KB row hit/miss) latency

Table 4.1: Simulator parameters for performance evaluation.

in, e.g., Sanchez et al. [95], or other optimization techniques such as iteration batching.

Profiling: To improve performance (in both Daedalus and baseline), we record serial and parallelized cycle counts for each loop, and parallelize only loops with $\geq 1\%$ speedup. Our simulation involves two passes: one to collect timing and another with only chosen loops parallelized. This is a proxy for more advanced compiler/runtime tuning heuristics (which are orthogonal to our work).

Baseline: We evaluate Daedalus against an affine-indexing-based array access analysis. This analysis can understand, e.g., that the array reference `array[i]` is distinct w.r.t. the loop with induction variable i, and that `array[2*i]` and `array[2*i + 1]` do not alias.

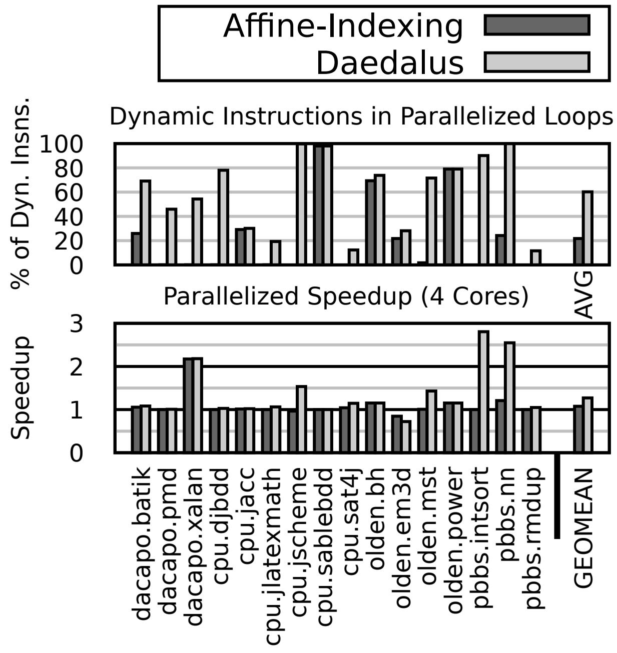
Analysis Workflow: We run Doop's *fact generator* on the benchmark, and our semantic-model compiler `modelc` on the models, encoding both into Datalog facts. We run a points-to analysis (with 1-object-sensitivity [73]). Then we run Daedalus, finding parallelizable loops. Datalog analyses are compiled with Soufflé [97, 56], with several optimizations.

Although semantic models usually improve precision, in a few cases an analysis without models performs better because our models are sometimes too conservative. We thus took the union of parallelizable loops from analyses with and without models enabled. (This is sound because one can prove a loop parallelizable by analyzing either the original code or the code with models substituted.)

Benchmarks: We evaluate Daedalus on the same 29 benchmarks as the previous chapter: DaCapo [23], JOlden [11] (a Java translation of Olden [27]), a Java implementation of portions of the Problem-Based Benchmark Suite [100], as well our *cpu* suite, a set of CPU-intensive programs individually chosen for complex control flow and data structures (in particular, compilers, parsers, and simulators). The *cpu* suite contains: circuit [44], CloudSim [26], DJBDD [68], Jacc [54], Janino [3], JGraphT [5] JLaTeXMath [6], JScheme [13], and raytracer [74].

4.6.2 Loop-Parallelization Speedup

We first evaluate the effectiveness of Daedalus at finding and exploiting parallelizable loops to improve performance. Fig. 4.8 shows a measure of *coverage*, or fraction of all dynamic instructions that are within any parallelizable loop (top), and the resulting speedup when these parallelizable loops are parallelized on



Excluded (< 5% coverage): dacapo.luindex, cpu.{cloudsim,jgrapht,jtidy},olden.{bisort, health,perimeter,treeadd,tsp,voronoi}, pbbs.{comparisonsort,convexhull,raycast}

Figure 4.8: Main results: coverage (parallelizable instructions) and parallel speedup.

4 cores (bottom), as measured via simulation (§4.6.1). This figure selects the *high-coverage* subset of our benchmarks: the 16 of 29 benchmarks total that have $\geq 5\%$ coverage under Daedalus. The low-coverage benchmarks are listed at the bottom of the figure.

Among these 16 benchmarks, 60.2% of all dynamic instructions occur within a parallelizable loop. In contrast, the baseline affine-indexing-based analysis finds parallelizable loops covering only 21.8% of instructions. Daedalus is more effective because it is able to derive distinctness facts, including for pointers that are stored in high-level data structures, and use these to resolve store aliasing questions.

As a result, Daedalus achieves a geomean speedup of 27.4% on 4 cores, compared to the baseline's 7.6%. While many array-based loops are parallelizable under both analyses, in 14 of the 16 benchmarks, additional speedup occurs because of increased loop coverage. The two best cases, `pbbs.intsort` (integer sort) and `pbbs.nn` (nearest-neighbors), achieve 2.8× and 2.5× speedup respectively, compared to no or very little baseline speedup. This is due to additional inferences made: e.g., in `pbbs.nn`, Daedalus is able to prove the independence of temporary objects in main loop iterations, including in nested and recursive calls, and of commutative stores into a shared results `HashMap`.

We evaluate scalability as well: on 16 cores, geomean speedup is 44.6%, compared to 8.3% in the baseline, with a best case of 6.3× in `pbbs.nn`. Thus although the discovered parallelism is often not perfectly scalable, as a regular scientific computation would be, there is still significant available speedup despite accesses to shared data structures.

4.6.3 Parameter Studies

Locking

With all locking removed, geomean performance on the benchmarks in Fig. 4.8, on a 16-core system (to stress locking), improves from 44.6% to 45.0%, an improvement of 0.4%. We thus conclude that though the inserted locking has some overhead, this overhead does not negate the benefits of parallelization. (Note that while this would not be sound in an actual program transform evaluation, we can simulate this and see the effects because the instruction trace pre-determines correct behavior.)

Workqueue Latency

So far, we have assumed a 10-cycle latency to distribute parallelized iterations, as we discussed in §4.6.1. By reducing this latency to 1 cycle (an unrealistic upper bound), we see a geomean speedup on a 16-core system of 51.7%, an improvement of 7.1%. On the other hand, increasing this latency to 100 cycles, speedup reduces to 7.6%. We thus conclude that job-spawn latency is an important metric to optimize in a high-performance parallel runtime.

4.6.4 Real-System Speedup

To provide confidence in the feasibility of achieving the simulated speedups, we parallelize `pbbs.nn` using a simple parallel runtime with an ordinary centralized workqueue. Averaged over 5 runs on a 4-core, 8-thread Haswell machine, we measured a speedup of 3.08×. This falls between our simulated 2.5× speedup on 4 cores and 4.0× on 8 cores.

4.6.5 Analysis Complexity and Runtime

Finally, we measured the runtime of Daedalus. Andersen points-to analysis has a worst-case complexity of $O(n^3)$, but often closer to $O(n^2)$ for realistic Java programs [102]. Map support makes complexity $O(n^4)$ in the worst case, because we also quantify over the key points-to set, but again in practice runtimes are reasonable because maps are not indexed by every abstraction in the program. Distinctness analysis inherits this complexity, with an additional factor for loop-nest size, which is usually shallow in practice. Over all benchmarks, a combined points-to and Daedalus analysis took an average of 0:23:26 (HH:MM:SS, geomean) on a 32-core machine (two-socket AMD Opteron 6272 at 2.1GHz), with a minimum of 0:12:10 (pbbs.comparisonsort) and maximum of 9:30:47 (dacapo.batik). (Analyses on two benchmarks, `cpu.cx3d` and `dacapo.fop`, timed out after 12 hours. These were excluded from results.) We believe further optimization opportunities are possible by tuning Datalog execution plans.

4.6.6 Case Studies

`nn`: Nearest-Neighbors Computation

The `pbbs.nn` benchmark performs a *nearest-neighbors* computation: given a list of points, for each point, it performs lookups in an octree (an efficient spatially-indexed tree) to find the k nearest points. Then, the list of nearest neighbors is inserted into a results map, indexed by the point itself.

Daedalus is able to determine that the outer loop over points is parallelizable: the points in the list are distinct, the octree lookup accesses only read-only data, and the write into the results map is safe because it is indexed by the distinct point. Only Daedalus, not the baseline array-based system, is able to deduce this parallelizability. This is for a few reasons: first, the per-point computation allocates some storage for intermediate results, and the baseline analysis is not able to determine that writes to this storage are iteration-local, because the storage is accessed through several levels of indirection (via fields on an object passed down through a recursive call-graph). Second, the writeback of the result to the map can only be seen as safe by combining (i) knowledge that each point in the iterated-over list is distinct, which we know by analyzing the loop that creates the list; and (ii) knowledge that the method call is a key-value map insertion, and such insertions are commutative if the keys are distinct. Thus, distinctness analysis on data-structure-aware IR is key to discovering `nn`'s potential for speedup.

`djbdd`: BDD Simplifier

The `cpu.djbdd` benchmark builds and then simplifies BDDs (binary decision diagrams), which are graphs that represent Boolean functions. To do this, it holds BDD nodes in a map, indexed by vertex ID. The main loop (by runtime) iterates over keys, fetching each node and performing some update. In order to parallelize this, an analysis would need to determine that each node in the map is distinct.

Unfortunately, Daedalus in its present state cannot derive this fact, and the reason illustrates a general difficulty with static analysis. The benchmark contains a loop that builds the BDD, allocating nodes one at a time, and Daedalus is able to infer that each node inserted at this point is distinct. However, the benchmark also implements logic operators on BDDs that may insert additional nodes. Each operator, when creating a new node, first checks if that node (uniquely identified by its content) is already present in the graph; if so, the operator re-uses the existing node, in order to de-duplicate the BDD. This logic foils the analysis because the analysis is not flow-sensitive: it must assume that a node could be added twice. In such cases, however, a *runtime* check could be inserted into transformed code so that the parallelized version is executed only if no duplicates exist. The following chapter describes one approach to inserting such runtime checks.

4.7 Related Work

Several other works have proposed variants of alias analysis to address the loop parallelization problem. Most recently, work by Johnson et al. [53] builds an alias analysis that derives conclusions of the form "`x` aliases `y` in an earlier/later iteration of loop L." This system works from the bottom up, resolving particular questions by posing sub-questions and attempting to prove them. In this way, they can combine many small, independent alias analyses, each encapsulating one particular rule, to form a robust ensemble of analyses capable of understanding many real programs. However, the core of their contribution is fundamentally the framework, and there is no high-level property of the program that they derive in a systematic way. In contrast, DAEDALUS is a principled top-down extension of Andersen points-to analysis and the points-to graph, refining its information for the whole program. In addition, we explicitly derive and make use of data-structure invariants such as the unique-value invariant in the key-value map described above.

At least one past work also refines the points-to graph by adding indexing to the abstraction: Wu et al. [114] propose an analysis analogous to the conventional array-access analysis for the purpose of loop parallelization. Though their analysis addresses its intended problem well, the approach quickly becomes untenable as program complexity grows. This is because in many programs, there is no closed-form expression for *which* object is visited each iteration, as we motivated earlier. Unlike this work's approach, we *can* often prove that each iteration visits a *different* object than all the others, even though we cannot express which one. The fact that this is sufficient for loop parallelization is the fundamental insight of DAEDALUS.

Earlier work by Wu and Padua [115] proposes a loop-parallelization system that explicitly understands arrays and hash-tables, as DAEDALUS does. In addition to the similarities to semantic models, which we noted already in §3.7, this system has some relevance to distinctness analysis because it also must resolve accesses to indexed data structures each iteration. However, unlike DAEDALUS, this system does not appear to have a general means of doing so, instead analyzing indexing and access conflicts directly and explicitly. Additionally, it must perform runtime checks to determine whether hash-table keys collide.

Past work has observed that understanding commutativity is key to uncovering additional parallelism [92, 82, 10, 62, 33]. This past work finds commutative loop iterations either by static analysis [92, 10] or by requiring explicit annotations over APIs [82, 109]. The use of commutativity is similar to DAEDALUS' modeling of commutative side-effects when analyzing parallelizable loops. The main difference between these past works and ours is simply the extent to which commutativity is the central contribution: in DAEDALUS, recognizing commutative side-effects helps to find more parallelizable loops, but it is orthogonal to our core contribution of distinctness analysis.

4.8 Chapter Summary

In this chapter, we introduced DAEDALUS, an analysis that computes *distinctness*. Distinctness is a specific type of (non-)aliasing between particular instances of pointers: for local variables, pointers in different iterations of a loop; for object fields, pointers in different instances of an object represented by one heap abstraction; for map values, pointers at different keys in the same map, or in all maps represented by a heap abstraction. We provided a set of inference rules that define a whole-program analysis to derive these distinctness facts. We then provided a set of conditions by which to evaluate loops in a program to verify whether they are parallelizable, using distinctness as the key means of resolving cross-iteration aliasing. Using this analysis, we demonstrated significant speedup in simulation on a number of real Java programs.

Chapter 5

Icarus: Extending Static Loop Parallelization Analysis with Dynamic Checks

A program was once in existence
Its instances remarkably consistent
Yet those few exceptions
Were utter deceptions
To static analysis, resistant.

So far, we have presented *static* analyses: these reason about the program in the abstract, producing conclusions that are true for all possible executions of the program. We have shown that the Daedalus static analysis discovers many parallelizable loops, and that parallelizing these loops results in significant speedup in the programs that contain them. However, a static analysis may not always be able to prove the facts that are needed to perform a desirable program transformation such as loop parallelization.

In this chapter, we describe a new system, Icarus (Integrated Compiler and Runtime with User-level Semantics), that improves the precision of the loop-parallelization analysis by using *dynamic checks* of necessary program invariants in addition to static reasoning. This hybrid system is able to verify that, for example, all items in a list are distinct just before iterating over that list with a parallel loop. Having verified that one dynamic fact, it can then make use of other facts that it has statically proven to be implied by the verified fact. When dynamic checks fail, the system resorts to a fully sound fallback strategy, preserving semantics in the uncommon case while enabling significant speedup overall. Importantly, this system is derived from the inference rules of Daedalus in a *principled* way: the original logic can be transformed into two additional analysis passes that interact with a client analysis (such as loop parallelization) in such a way that only the necessary dynamic checks are inserted. This approach serves as a template for future hybrid static/dynamic systems.

We introduce Icarus in this chapter, including inference rules, but we do not include all details of the soundness proof; that proof is included in Appendix B.

5.1 Motivation: Almost-Provable Analysis Facts

Though static analyses can be engineered to be remarkably precise, they sometimes fall short of proving what is needed. This can occur for two main reasons.

The first reason is that the analysis is *imprecise*: while the conclusions that it produces are always true, it may not necessarily derive all conclusions that are true. This is due to the undecidability of the Halting Problem [108], and thus values during program execution in general. For example, a loop may contain pointers, array indices or map keys computed by an arbitrarily complex function, and its parallelizability may depend on whether or not the resulting memory accesses overlap. No analysis can provide an answer to this question in all cases.

The second reason is that analysis results must be true for *all possible* program executions. For example, Daedalus must only claim that a local variable is distinct with respect to a loop if this is true for all program inputs. However, there are programs where distinctness is *usually* true, but this is not necessarily guaranteed by the program itself. For example, a program might read a data structure from a file and iterate through its elements. If the loop iterations are always independent for a "well-constructed" input, it might be desirable to parallelize this loop: otherwise, the system is missing a significant opportunity for speedup. But it would actually be unsound to parallelize the loop statically. The decision to perform work in parallel can only be made at runtime, once the program has observed the input file.

To see an example of this, consider the program in Fig. 5.1. This program processes `Item` objects, storing them into a list then performing some sort of computation over them. Assume that some external caller calls `add()` multiple times with different arguments, possibly passing the same `Item` instance more than once. The logic in `add()` ensures that each item is added to the list only once. (The logic here is somewhat simplistic, but similar deduplication logic occurs in real programs; for example, in DJBDD [68], BDD nodes are deduplicated on insertion.) Thus, the loop at line 12 sees each `Item` exactly once: `it` is *distinct* with respect to this loop. The computation in `compute()` is thus completely independent per iteration, and the loop at line 12 should be parallelizable.

Unfortunately, Daedalus as posed cannot deduce this distinctness fact. In principle, it is possible to develop an inference rule that pattern-matches the deduplication conditional at line 5 and manually injects a distinctness fact for the list. However, such an approach is ultimately as futile as any other pattern-matching approach to the Halting Problem: we cannot chase every possible program design with a new inference rule.

```
1  ArrayList<Item> list;
2  HashMap<Item, Metadata> metadata;
3
4  void add(Item it) {
5    if (!it.seen) {
6      list.add(it);
7      it.seen = true;
8    }
9  }
10
11 void process() {
12   for (Item it : list) {
13     it.result = compute(it);
14   }
15 }
16
17 int compute(Item it) {
18   Metadata m = metadata.get(it);
19   m.update();
20   return m.result();
21 }
```

Figure 5.1: Example of a program where a static analysis, Daedalus, can *almost* but not quite prove the facts needed to parallelize the loop. The analysis cannot prove that the items in `list` are distinct because it does not understand the deduplication logic on list insertion at lines 5–8. Thus, the loop at line 12 is not parallelizable, even though it should be.

Let us reconsider Fig. 5.1 in light of other possible approaches. If we take for granted that the contents of `list` are a black box – they may or may not be distinct, as far as the analysis knows – then a natural approach is to *check* for the properties that we need, and then behave as appropriate for this situation. If a program can dynamically determine whether the elements in `list` are distinct for any *particular* call to `process()`, then for that particular instance, it can branch to either a parallelized version of the loop at line 12 (if distinct) or the original serial version (if not).

A naïve approach to extending precision with dynamic checks might be to insert a check wherever a distinctness fact is needed for parallelization. However, this quickly becomes impractical. In this example, there are several writes to the heap within the loop iteration. The write at line 13, directly to a field of `it`, is clearly parallelization-safe if `it` is checked to be distinct. However, consider the access at line 18 to the `metadata` key-value map. Assuming that the contents of `metadata` are within-map distinct, then `m` should also be distinct, allowing the mutation inside `m.update()` to be parallelization-safe. But the analysis will not have this distinctness fact, so the naïve approach inserts a distinctness check on `m` too. This check is, however, redundant: if we have already checked that `it` is distinct, then we know `m` is too. In other words, we can *statically* derive *conditional* distinctness facts, and use these to cascade one dynamic check into a large number of useful conclusions.

We observe that dynamic checks for distinctness in particular can be relatively cheap: simply a hash-table lookup in a hash table keyed on values that have already been seen in this loop instance, and then insertion if the lookup fails. Thus, if these checks enable sufficiently beneficial program transforms, then the use of dynamic checks could yield significant benefit for a relatively small cost. In the remainder of this chapter, we outline a general approach to extend a static analysis with such dynamic checks.

5.2 One Solution: Fully-Dynamic Version of a Static Analysis

One initial plausible approach to this problem is to convert the static analysis directly to a dynamic analysis. This can be done via a direct correspondence between static abstractions of values (program variables and heap objects) and dynamic instances of these values, evaluating rules in real time where applicable. Then, the program would have available, at runtime, a precise set of facts tailored to the particular execution so far.

5.2.1 Building a Dynamic Analysis

In order to perform such a transform, let us begin with the static analysis and categorize the *relations*, or logic predicates, such as $\text{NotDistinct}(v, L)$, into *value-attached* predicates and *other* predicates. The value-attached predicates describe a particular value (dynamically), or some abstraction for a set of values (statically). The other predicates describe the program in some way that is usually truly static and independent of its particular execution: for example, the loop structure. (Some predicates may differ in various executions but not be directly value-attached; we consider them to be in the *other* category in this case.)

We must design a way for predicates to be *dynamically* attached to values in the program dataflow. Such a system is often known as a Dynamic Information-Flow Tracking (DIFT) analysis, and is commonly used for applications such as security-related taint tracking. Efficient design of such a system is beyond the scope of this work (as we are merely posing a hypothetical design point).

Then, for all inference rules, we find those that generate a value-related predicate (in the consequent, i.e., under the horizontal line), and for these, insert code into the transformed program that, at the appropriate points, determines whether the rule applies. Usually, rules are written in a way that corresponds directly to program structure and dataflow, so that properties of the output of a particular statement depend only on the properties of that statement's input(s), or perhaps a heap object on which the statement acts. Adaptation of an inference rule to such a case is straightforward.

Such a system could indeed generate distinctness facts dynamically, and these facts would be potentially much more precise than those produced by the purely static system in Chapter 4. However, it suffers from two major problems, which we now describe.

5.2.2 Problem 1: Performance

The first major problem with a *purely* dynamic system is performance. As we motivated in §5.1 above, it is often the case that a single dynamic observation can lead to a number of additional facts statically. In other words, if we can statically determine that distinctness of variable a implies distinctness of variable b, then we need only check a; we do not need to separately check b. But the purely dynamic system works *only* by dynamic application of inference rules, and so performs significantly more work than an analysis that could do some pre-computation statically.

5.2.3 Problem 2: Correctness

The second major problem with a dynamic system is that a dynamic fact about the program does not carry the same meaning as a static program fact, as we describe below. As a result, it is not immediately clear how the compiler can use a dynamic fact to, e.g., justify a loop parallelization decision.

Limited Scope of Dynamic Fact

A dynamically-checked fact differs from a statically-derived fact in several ways. First, while a statically-derived fact must apply to *all* executions, a dynamically-checked fact necessarily only applies to the current execution. Fortunately, this limitation is not a problem as long as the dynamically-checked fact is only used to justify decisions for the current execution.

Second, however, and more critically, while a statically-derived fact generally applies to a variable or program point for *all* time during the execution, a dynamically-checked fact is only necessarily true at the moment that it is checked. If a fact that has been verified in the present was not true in the past, we might erroneously rely on assumptions that depend on the fact having *always* been true. For example, say that we rely on a non-aliasing fact that shows a particular value in memory has not changed under a certain pointer. A dynamically-checked aliasing fact may show that another stored-to pointer does not alias this pointer *now*, but the other pointer may have aliased this one in a prior iteration of some loop, and so the value may already have been overwritten. Likewise, if the presently-verified fact may not be true in the future, then the program might make a decision (such as branching to a parallelized version of a loop) that will later lead to unsoundness when the dynamically-checked invariant is violated.

Possibility of Check Failure

Finally, unlike statically-derived facts, dynamically-checked facts *may not be true* in any given execution. Unless it is valid for the program to simply abort (i.e., the user allows "correct answer or explicit failure" semantics), the program must have some fallback path that will also perform the correct computation. This is not difficult if a check can be performed prior to any execution of transformed code: the program can simply branch to the original version of the code (e.g., the non-parallelized loop) if the check fails. However, failure paths become significantly more complex when failures are detected partway through transformed code's execution, after side-effects have occurred.

As a result of these downsides to a purely-dynamic analysis, and these challenges posed by dynamic facts, we propose a *hybrid* static-dynamic system, and we reason carefully about the dynamic scope of facts that are derived dynamically. We now describe this approach.

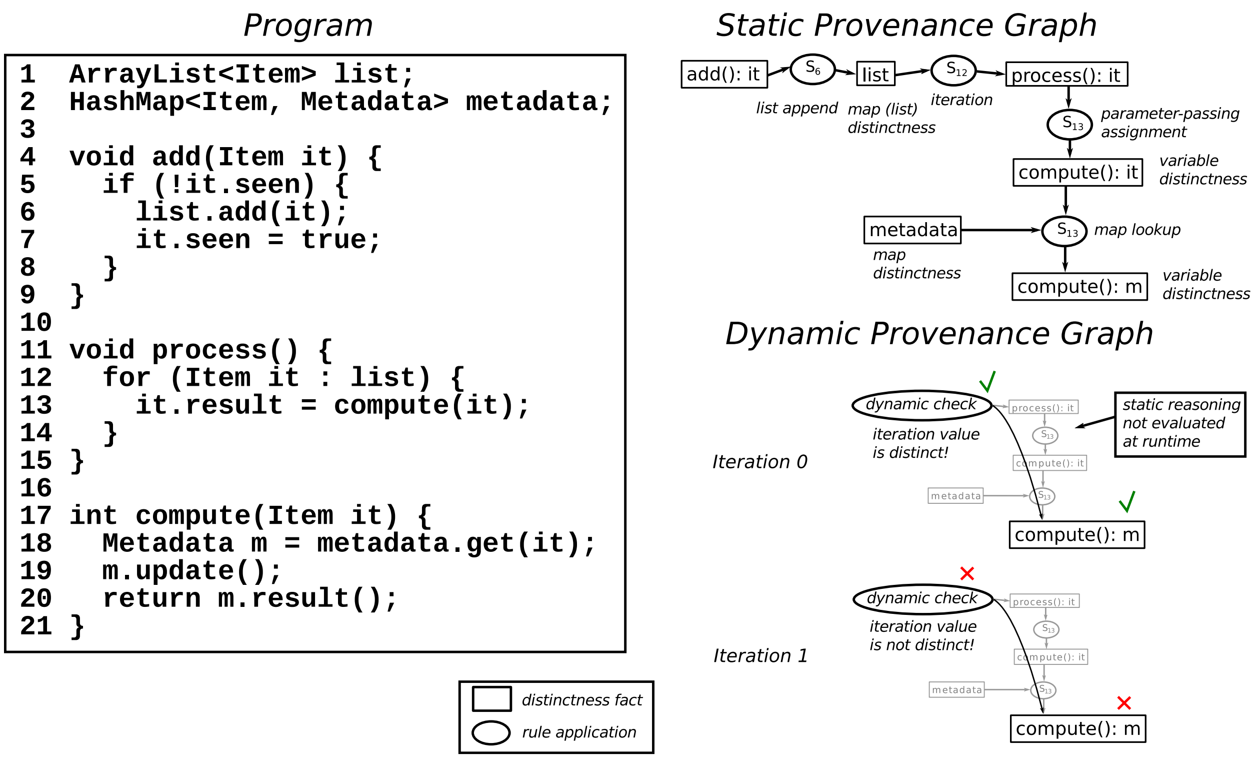
5.3 Our Approach: Hybrid Static-Dynamic System

In order to combine the precision of dynamic analysis with the low overhead of static analysis, we propose a *hybrid* static-dynamic system, Icarus. This system extends Daedalus in a principled way with dynamic checks of variable distinctness (§5.4.1) and propagation of not-distinct taints through object fields (§5.4.2) where necessary in order to parallelize many more loops.

The hybrid scheme's key goal is to perform dynamic checks that are at the *root* of potentially several needed conclusions: for example, by showing that the pointer loaded at the start of loop iteration is distinct, many other distinctness facts follow through the normal application of static-analysis inference rules, leading to many static facts that are conditionally true based on a particular check. The analysis is capable of finding these conditional implications during static analysis.

Fig. 5.2 illustrates this hybrid scheme in more detail. For the same example program as in Fig. 5.1, we illustrate the system's reasoning using *provenance graphs*. A provenance graph is simply a representation of the chain of reasoning to arrive at a fact or facts with inference rules. The static provenance graph on the upper-right shows the series of rule applications that would allow us to infer several essential facts, such as the distinctness of `it` at line 12 in `process()` and `m` at line 18 in *compute()*, that would allow us to parallelize the loop at line 12. However, as we discussed earlier, the existing Daedalus system cannot infer that the `add()` method adds a distinct `it` to `list` on each call, so this chain of inferences fails at the first step.

However, if the system is able to decide to check certain facts at runtime instead, then it can simply designate some fact along this chain as dynamically-checked, and then use the rest of the series of inference steps to arrive at the needed facts, contingent on a successful dynamic check. The dynamic provenance graph in the lower-right shows how a series of dynamic facts, which are particular instances of the desired static facts, are inferred with the dynamic application of rules. The key aspect of the hybrid system that differentiates it from a purely dynamic system is that the *intermediate reasoning remains static*: only the dynamic check is needed, and this directly implies (at runtime) the distinctness of `m`. We will see how this is actually implemented later in §5.3.3.



```
1  ArrayList<Item> list;
2  HashMap<Item, Metadata> metadata;
3
4  void add(Item it) {
5    if (!it.seen) {
6      list.add(it);
7      it.seen = true;
8    }
9  }
10
11 void process() {
12   for (Item it : list) {
13     it.result = compute(it);
14   }
15 }
16
17 int compute(Item it) {
18   Metadata m = metadata.get(it);
19   m.update();
20   return m.result();
21 }
```

Figure 5.2: Example operation of a *hybrid static-dynamic analysis*, at a high level. The provenance graphs on the right represent the chain of reasoning to obtain certain distinctness facts. When some of these facts cannot be proven statically, the system might choose to dynamically check (e.g.) the distinctness of `it` in the loop at line 12. The remainder of the inference chain is still statically valid, so this dynamic check leads directly to a needed conclusion, the distinctness of `m` at line 18, without going through the intermediate rule applications for each dynamic instance. In this case, the compiler could parallelize the loop at line 12 (with stores at lines 13, 19, and 20) with just this one dynamic check.

A key insight to our approach is that the analysis that determines which dynamic checks are needed, and that performs the static analysis that is based on the hypothetical success of these checks, can actually be a derivative of the original inference rules: we can follow potential inference chains (i.e., proof trees) backward statically, and fill in the missing pieces with dynamic checks, using inference rules that are themselves constructed in a mechanical way from the originals. We must be careful in this process, however, to (i) only perform transforms based on facts that can actually be checked, and (ii) minimize the configuration-space search overhead in determining where to place checks. The key step to achieve this efficiency is the use of a *two-pass design* where (i) all possible dynamically-checked facts are derived at once, without any information about how to prove them (so that the many different proof options are merged, reducing overhead at this stage), and then (ii) the client analysis indicates which facts are actually needed, and only at this point do we perform the backward traversal to insert checks. This two-pass approach is illustrated in Fig. 5.3.

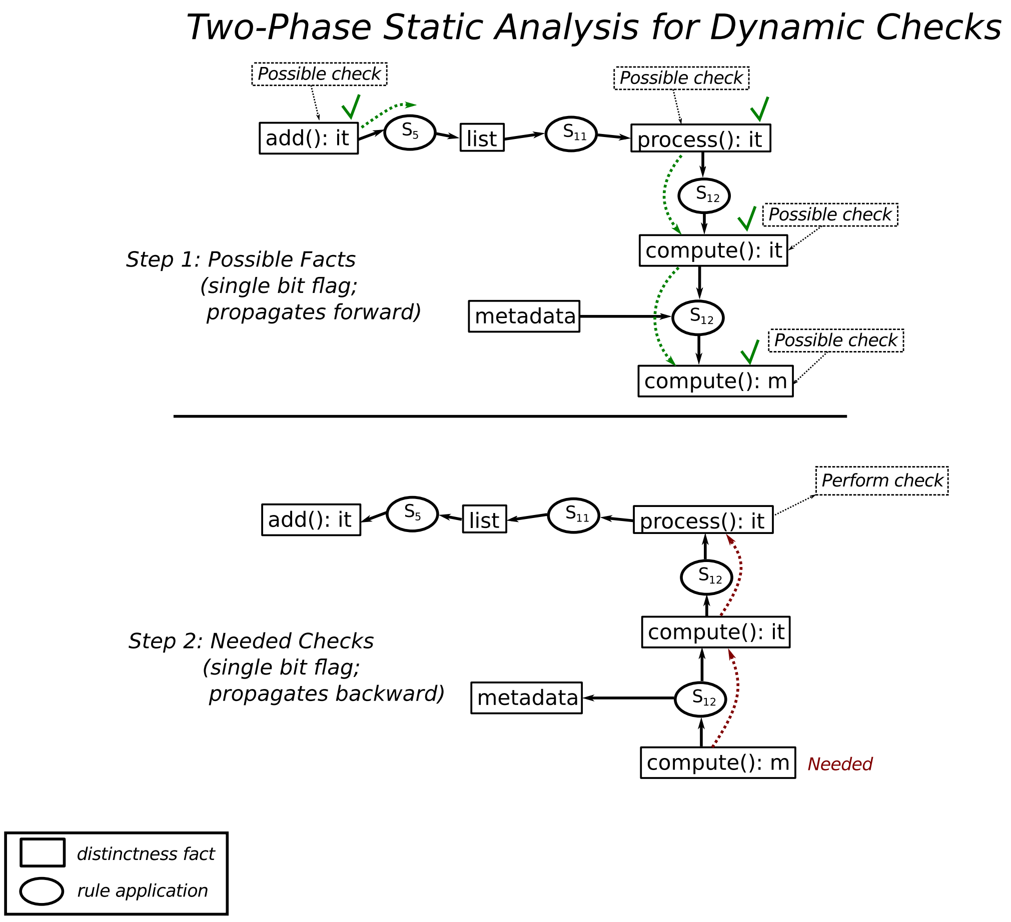

Two-Phase Static Analysis for Dynamic Checks
Possible check
add(): it
S5
list
S11
process(): it
S12
compute(): it
metadata
compute(): m
Step 1: Possible Facts (single bit flag; propagates forward)
Perform check
Step 2: Needed Checks (single bit flag; propagates backward)
Needed
distinctness fact
rule application

Figure 5.3: The hybrid static-dynamic analysis is derived from the original static analysis by mechanically constructing two new phases: (i) possible facts, which are propagated forward (conceptually, in the provenance graph) through inference rules in the same way as static facts; and (ii) needed facts, which are propagated backward until they are satisfied by actually inserted dynamic checks.

5.3.1 Computing Possible Distinctness

The first analysis pass in the hybrid static-dynamic system computes *all possible* distinctness facts that could be dynamically proven. This is necessary to compute first in order to constrain the later solution-space search: otherwise, a client analysis such as loop parallelization may decide to parallelize a loop and then request a dynamic check for some variable's distinctness that either (i) can be statically shown to never succeed or (ii) cannot be done in the transformed code for some technical reason, such as that the variable is a synthetic temporary or the program point is inside a synthetic model.

The possible-distinctness facts are represented as negations (to simplify the rules, due to the way that they join at meet-points), just as static distinctness facts are. The facts are represented by the judgments NotPossibleDistinct(x, L) and FieldNotPossibleDistinct$(A.f)$.

In order to determine this set of possible facts, the analysis begins by annotating every variable whose distinctness it can *directly* check. While we will cover the detailed design of a runtime distinctness check in §5.4.1, it suffices for now to assume that any value whose machine-code (or JVM bytecode)-level register allocation is known can be checked directly. In Icarus, this corresponds to every SSA variable (language-level local or temporary) in every method that is not part of a semantic model. The analysis thus initializes the set of possible facts with distinctness facts containing these variables and all loops in context at the variable definition site, as shown in Rule 23.

Note, in addition, that only *variable* distinctness is directly checked in the Icarus system. In contrast, field distinctness and map distinctness cannot be checked directly. This is due to a key difference in definitions: variable distinctness is defined in terms of values in iterations of a loop, which we can directly observe at the location of the inserted check as execution proceeds, while field distinctness is defined in terms of *all* objects represented by a given heap abstraction. The only way to check this property, from first principles, would be to dynamically track the corresponding heap abstraction for every concrete object, and iterate over all such objects to check field values. This would impose a large runtime cost both due to object tracking and the check itself. Thus, only variable checks are "possible" at this stage of the analysis.

Rule 23. *It is possible to check the distinctness of any variable for which we have a JVM bytecode offset and stack slot number or local variable slot number.*

$$[\textsc{PossibleCheck}]\ \frac{[x_i := \ldots]_i \qquad \mathit{BytecodeOffset}(S_i, \_) \qquad \mathit{StackSlot}(x_i, \_) \vee \mathit{LocalVar}(x_i, \_)}{\mathit{CheckPossible}(x_i)}$$

Rule 24. *If a check is possible at a local variable definition, or else if the local variable has been statically proven distinct, then that variable is possibly distinct. Possible-distinctness of assignments and loads is handled separately. Thus, a variable is not possibly distinct if no check is possible, it is statically distinct, and it is not the result of an assignment or load.*

$$[\textsc{PossibleCheckDistinct}]\ \frac{\begin{array}{ll} [x_i := \ldots]_i & \\ \neg[x_i := \phi(\ldots)]_i & \neg\mathit{CheckPossible}(x_i) \\ \neg[x_i := y_i.f] & \mathit{NotDistinct}(x_i, L) \end{array}}{\mathit{NotPossibleDistinct}(x_i, L)}$$

The analysis also immediately rules out distinctness checks on any variables known to be constant (Rule 25). This avoids needless effort attempting to prove parallelizability and then dynamically check it at runtime when it will never succeed.

Rule 25. *Any variable statically proven to be constant w.r.t. a loop is not a possibly-distinct variable, because no dynamic check will ever succeed.*

$$[\textsc{NotPossibleConstant}]\ \frac{[x_i := \ldots]_i \qquad L \in \mathbb{L}(S_i) \qquad \neg \mathit{NotConstant}(x_i, L)}{\mathit{NotPossibleDistinct}(x_i, L)}$$

Next, the analysis propagates possible-distinctness facts in the same way that it propagates ordinary distinctness facts. The rules to perform this analysis are mechanically derived from the original rules by replacing $NotDistinct(v, L)$ with $NotPossibleDistinct(v, L)$ in the consequent (or the conjunction of the two in the antecedent) and $FieldNotDistinct(A.f)$ with $FieldNotPossibleDistinct(A.f)$ in the consequent (likewise the conjunction if in the antecedent).

To demonstrate this simple transform, we show the rules for possible-distinctness propagation for assignment, loads and stores here.

The rule to propagate possible-distinctness through assignments (single or multiple input) is shown in Rule 26. This rule is as simple as the one for ordinary static distinctness: not-possible facts are propagated from sources to destinations (or equivalently, the output is possible-distinct if all inputs are possible-distinct). A local check also results in possible distinctness.

Rule 26. *If all inputs to an assignment are possibly-distinct, then the output is possibly-distinct too. Thus, if any input to an assignment is not possibly distinct or statically proven distinct, and if no local check is possible, then the assignment result is not possibly distinct.*

$$[\textsc{PossibleAssign}]\ \frac{\begin{array}{ll} & \mathit{NotDistinct}(x_i, L) \\ & \mathit{NotPossibleDistinct}(x_i, L) \\ [x := \phi(x_1, \ldots, x_n)]_i & \neg \mathit{CheckPossible}(x) \end{array}}{\mathit{NotPossibleDistinct}(x, L)}$$

Next, the rules to propagate possible-distinctness through loads are shown in Rules 27 and 28. The rules propagate possible-distinctness as long as both the base pointer is possibly (or statically) distinct, and the field in all pointed-to abstractions is possibly (or statically) distinct.

Rule 27. *If a load's base pointer is not statically distinct, and not possibly distinct, then the result is not possibly distinct.*

$$[\textsc{PossibleLoadBase}]\ \frac{[x := y.f]_i \qquad \mathit{NotDistinct}(y, L) \qquad \mathit{NotPossibleDistinct}(y, L)}{\mathit{NotPossibleDistinct}(x, L)}$$

Rule 28. *If the field in a load's pointed-to abstraction is not statically distinct, and not possibly distinct, then the load result is not possibly distinct.*

$$[\textsc{PossibleLoadField}]\ \frac{\begin{array}{ll} [x := y.f]_i & \mathit{FieldNotDistinct}(A.f) \\ A \in \mathit{pts}(y) & \mathit{FieldNotPossibleDistinct}(A.f) \end{array}}{\mathit{NotPossibleDistinct}(x, L)}$$

The possible-distinctness rules for field stores are given in Rules 29 and 30. These rules mark a field possibly-distinct as long as no store overlap occurs (if it does, even our dynamic-tracking approach described later in this chapter cannot dynamically prove the field to be distinct), and as long as the stored value is statically or possibly distinct if the pointer is not constant w.r.t. a given loop.

Rule 29. *If a store overlaps with another store to a particular abstraction and field, then the field is not possibly distinct.*

$$[\textsc{PossibleStoreOverlap}]\ \frac{\begin{array}{ll} [x.f := y]_i & \\ [x'.f := y']_{i'} & X \in \mathit{pts}(x) \\ i \neq i' & X \in \mathit{pts}(x') \end{array}}{\mathit{FieldNotPossibleDistinct}(A.f)}$$

Rule 30. *If for the one store to an abstraction and field, every stored value is statically or possibly distinct w.r.t. every loop in which the corresponding pointer is not constant, then the field is possibly distinct. Thus, if for this abstraction and for any loop, the pointer is not constant and the stored value is not statically distinct or possibly distinct, then the field is not possibly distinct.*

$$[\textsc{PossibleStoreValue}]\ \frac{\begin{array}{ll} [x.f := y]_i & \\ \mathit{NotConstant}(y, L) & \mathit{NotDistinct}(y, L) \\ A \in \mathit{pts}(y) & \mathit{NotPossibleDistinct}(y, L) \end{array}}{\mathit{FieldNotPossibleDistinct}(A.f)}$$

Importantly, at this stage of the analysis, only a *single bit* indicating possible distinctness is computed per variable per loop. The possible-distinctness information does not contain any *provenance* information, or in other words, it does not indicate *how* a particular value could be proven distinct or what checks would be necessary to do so. Computing this information upfront would both be prohibitively expensive, because a particular value might be proven distinct in many different ways, and unnecessary, because the use of possible-distinctness (as we will see in §5.3.2 below) is simply to allow parallelization logic (or other client analysis) to make decisions and select which distinctness facts it actually needs. The client analysis will likely need only a small fraction of all possible distinctness facts, so the specific check strategies are computed only then.

Finally, because the client analysis relies upon possible-distinct facts to make its decisions, the analysis *must be able to actually prove* what it says it can prove, with an appropriate set of checks. In other words, the word "possible" in this predicate should not be interpreted as meaning that a distinctness fact might be provable, and later stages might look into it further; rather, we *know* that we can prove it with a dynamic check, so we can commit to (e.g.) a loop parallelization based on this possible fact, with the knowledge that it will be possible to wrap this transformed loop with checks of some sort to retain soundness.

5.3.2 Loop Parallelization and Needed Distinctness Facts

Next, given that we have a set of *possible* distinctness facts that includes both the static distinctness facts and also the dynamic distinctness facts, we adapt the client analysis (the analysis that uses distinctness facts: in Icarus, the loop parallelization logic) to make use of these facts as if they were always true, and then determine which facts it actually needed to come to its final conclusions.

In the case of loop parallelization, this is a relatively simple adaptation. As described in §4.5, the main function of loop parallelizability logic is to detect *conflicting accesses*: any write within a loop body to a pointer that is not distinct with respect to that loop, or any pair of accesses (at least one of them a write) that are both distinct but are not proven to alias by the must-alias analysis. The changes to the parallelization logic in Icarus, as compared to Daedalus, consist of (i) using possible-distinctness facts instead of ordinary distinctness facts, (ii) ignoring tag conflicts, because these will result in static non-distinctness and thus dynamic checks, ensuring dynamic-check coverage in any case; and (iii) a second stage, *after* the final determination of which loops to parallelize, in which all distinctness facts for all written-to pointers in a loop body are gathered again, and those for which no static distinctness fact exists are noted to be *required dynamically-checked facts*. These are then fed back into the distinctness analysis for its second *need* phase.

5.3.3 Satisfying Needed Distinctness Facts

Finally, with the needed distinctness facts outlined by the client analysis, the *needed distinctness* pass operates, working backward over the provenance graph to determine where dynamic checks must occur.

Like possible-distinctness, needed-distinctness is propagated by a set of rules that are mechanically derived from the original static analysis's inference rules. Unlike possible-distinctness, this transform involves some choice by the analysis designer at several points to pin certain decisions and avoid backtracking, as we will see.

It is easiest to think of each needed-distinctness rule as following a general pattern: for an original distinctness rule that infers distinctness of a particular output (e.g., variable or field) from a set of input(s),

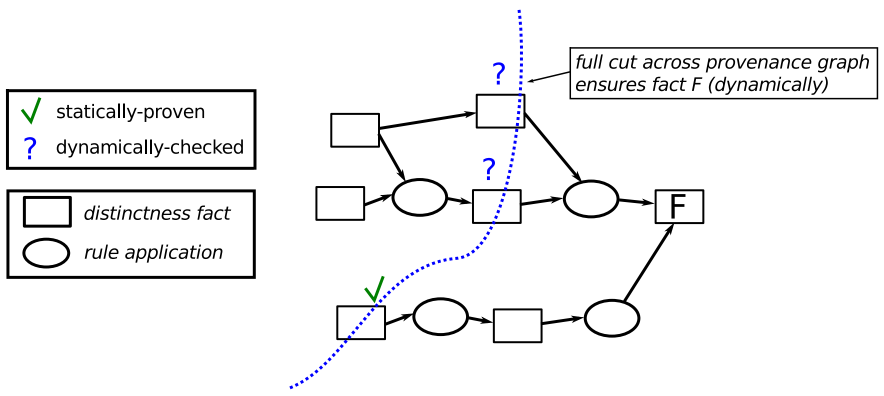


Figure 5.4: One way of seeing the hybrid static-dynamic approach is that dynamic checks are inserted to fill in gaps in a full cut across the static provenance graph from root facts to a desired fact F (needed for, say, loop parallelization).

the derived rule translates needed-distinctness on the output to one of (i) nothing, if the output is already statically proven distinct, (ii) a set of needed distinctness facts on the input(s), if those inputs have possible-distinctness facts, or (iii) if the need cannot be propagated backward, and a local check is possible, then a local check. One of these three options *must* apply as long as the client analysis only asserts needed distinctness on entities that have possible-distinctness facts, by construction: a possible-distinctness fact can only be injected by Rule 23 when a local check is possible (case (iii)) or by a possible-distinctness derived directly from an original distinctness rule and possible-distinctness on all required inputs (case (ii)).

Another way of seeing this principle is that the need-propagation ensures that a *cut* across the static provenance graph exists, covering all paths from root facts (those with no inbound edges) to the desired fact, where the cut consists of both statically-proven distinctness facts and dynamically-checked facts. Fig. 5.4 illustrates this view.

One notable aspect of needed-distinctness rules is that, unlike the static distinctness rules and possible-distinctness rules, the needed-distinctness rules operate with *positive* distinctness (i.e., they are not negated as the first two are), in order to match with the positive sense of dynamic checks and of requests from the client analysis. Thus, when we say that the needed-distinctness rules are derived from the original distinctness rules, we mean that they are derived from *inverted* versions of the original rules. This, too, is a mechanical transform using DeMorgan's Law.

Deriving Needed-Distinctness Rules

Before describing concrete needed-distinctness rules, we briefly discuss a set of general transforms shown in Fig. 5.5. These transforms demonstrate how to translate a distinctness rule that is composed of smaller predicates joined by AND- and OR-conjunctions into the corresponding needed-distinctness rule.

Original Distinctness Rule — *Transformed Needed-Distinctness Rule*

Conjunction rule, propagating:

$$\frac{A \quad B}{C}\ [\textsc{Orig}] \qquad \frac{\text{Need}(C) \quad \neg C \quad \text{Possible}(A) \quad \text{Possible}(B)}{\text{Need}(A) \quad \text{Need}(B)}\ [\textsc{Transformed}]$$

When the output of a rule is needed-distinct, and showing distinctness of that output ordinarily requires two other distinctness facts, then propagate need backward to both if both are possible-distinct.

Conjunction rule, local check:

$$\frac{A \quad B}{C}\ [\textsc{Orig}] \qquad \frac{\begin{array}{ll} & \neg C \\ \text{Need}(C) & \neg((\text{Possible}(A) \vee A) \wedge \\ \text{CheckPossible}(C) & \quad (\text{Possible}(B) \vee B)) \end{array}}{\text{Check}(C)}\ [\textsc{Transformed}]$$

When the output of a rule is needed-distinct, and showing distinctness of that output ordinarily requires two other distinctness facts, but one or both of those facts cannot be statically or dynamically checked distinct, then insert a local check.

Disjunction rule, propagating to first:

$$\frac{A \vee B}{C}\ [\textsc{Orig}] \qquad \frac{\begin{array}{lll} & & \text{Possible}(A) \\ \text{Need}(C) & \neg C & (\text{Prefer}(A) \vee \neg\text{Possible}(B)) \end{array}}{\text{Need}(A)}\ [\textsc{Transformed}]$$

When the output of a rule is needed-distinct, and showing distinctness of that output ordinarily requires one of two other distinctness facts, and either only the first is possible or both are possible but the first is preferred, then propagate need to the first. The preference logic is arbitrary and is used to choose a proof path so that no backtracking is needed.

Disjunction rule, propagating to second:

$$\frac{A \vee B}{C}\ [\textsc{Orig}] \qquad \frac{\begin{array}{lll} & & \text{Possible}(B) \\ \text{Need}(C) & \neg C & (\text{Prefer}(B) \vee \neg\text{Possible}(A)) \end{array}}{\text{Need}(B)}\ [\textsc{Transformed}]$$

Symmetric to the above. Separate rules for the first-input and second-input propagation cases are needed.

Disjunction rule, local check:

$$\frac{A \vee B}{C}\ [\textsc{Orig}] \qquad \frac{\begin{array}{ll} & \neg C \\ \text{Need}(C) & \neg(\text{Possible}(A) \vee A) \vee \\ \text{CheckPossible}(C) & \quad \neg(\text{Possible}(B) \vee B) \end{array}}{\text{Check}(C)}\ [\textsc{Transformed}]$$

When the output of a rule is needed-distinct, and showing distinctness of that output ordinarily requires one of two other distinctness facts, but neither fact can be statically or dynamically checked distinct, then insert a local check.

Negation rule:

$$\frac{\neg A}{B}\ [\textsc{Orig}]$$

(never occurs)

Negations never occur in the corecursive core rules of a monotonic analysis; thus, there is no need to specify a transform for negation operators in rules here.

Figure 5.5: Transforms to derive needed-distinctness rules from the original static-analysis distinctness rules. These needed-distinctness rules propagate needed-distinctness backward from requested dynamic-distinctness facts to determine where checks are necessary.

One notable detail in Fig. 5.5 is the handling of disjunctions (OR-rules). It is sometimes the case that proving a distinctness fact can be done in multiple ways, any of which suffices. For example, to show that the result of a `mapget`[1] instruction is distinct, it suffices either to show that the map abstraction is within-map distinct and the key variable is distinct (given that the map variable is constant) in the given loop, or to show the map abstraction is globally distinct and at least one of the key or map variable is distinct in the given loop. In order to avoid the need to backtrack in our search for an appropriate check strategy, we transform rules to statically prefer one option over the other. (The preference logic may be arbitrarily complex, but it must depend only on static attributes of the instruction and inputs to the needed-distinctness phase.) In the `mapget` case, a sensible heuristic is to disambiguate on the constantness of the map argument: if it is constant in the given loop, then one could propagate need only to within-map distinctness on the map/key combination (because within-map distinctness is weaker than global distinctness, thus likely cheaper to check) and distinctness on the key. Otherwise, one could propagate need to global-map distinctness and the key, unless the key has no possible-distinct fact, then to the map.

Note that if any means of dynamically proving distinctness is possible for the result, the needed-distinctness rules *will* find a way to prove it. The preference logic merely steers the choice of where to place checks in a likely-cheaper direction. The alternative, backtracking-based search, would be considerably more expensive, and would likely need an accurate dynamic-check cost heuristic in any case to succeed at optimizing overhead.

In addition, the propagate-before-local-check heuristic is designed to reduce check overheads, because it prefers pushing checks as far "up" (toward roots of value fan-out) as possible, so that one check on an initial value can be shared by many derived distinctness facts.

Needed-Distinctness Rules

We now describe several inference rules that form the core of the needed-distinctness analysis. These rules propagate needed-distinctness across assignment, load, and store statements.

First, Rules 31, 32 and 33 implement the backward propagation of needed-distinctness, or local check insertion if appropriate, for multiple-input assignments (ϕ-nodes). These rules implement three cases. First, if all inputs to an assignment can be proven distinct upstream (possible-distinctness facts), and there is no tag conflict producing a non-distinctness fact (see Fig. A.2 in the appendix describing Daedalus' inference rules), then we can propagate the needed-distinctness upstream, pushing checks as early as possible. However, if either some input cannot be checked at an earlier point, or else a tag conflict is causing non-distinctness

[1] Note that in the evaluated version of Icarus, we disabled the map dynamic-distinctness logic to reduce analysis overhead. However, in principle, there is nothing preventing dynamic distinctness from extending to any data structure or set of inference rules.

at this point no matter what values arrive on the inputs, then we must perform the check locally.

Rule 31. *If any input to an assignment is not statically distinct and not possible-distinct, then we will not be able to propagate needed-distinctness upstream.*

$$[\textsc{AssignInputNotPossible}]\ \frac{[v := \phi(v_1, v_2, \ldots, v_k)] \quad 1 \leq i \leq k \quad \mathit{NotPossibleDistinct}(v_i, L)}{\mathit{AssignInputNotPossible}(v, L)}$$

Rule 32. *If distinctness is needed on the output of the assignment, and it is not statically distinct already, and if no input is not statically or possibly distinct, and there is no tag-conflict-induced non-distinctness (§A.3), then propagate the need to inputs.*

$$[\textsc{AssignNeedDistinctPropagate}]\ \frac{\begin{array}{l}[v := \phi(v_1, v_2, \ldots, v_k)] \\ 1 \leq i \leq k \\ \mathit{NeedDistinct}(v, L) \\ \mathit{NotDistinct}(v, L) \\ |\mathit{Tag}(v)| = 1 \\ \neg \mathit{AssignInputNotPossible}(v, L)\end{array}}{\mathit{NeedDistinct}(v_i, L)}$$

Rule 33. *If distinctness is needed on the output of the assignment, and it is not statically distinct already, but if either some input has no possible distinctness fact or else there is a local tag conflict, then insert a check on the assignment result.*

$$[\textsc{AssignDistinctCheck}]\ \frac{\begin{array}{ll}[v := \phi(v_1, v_2, \ldots, v_k)] & \\ \mathit{NeedDistinct}(v, L) & |\mathit{Tag}(v)| > 1 \ \vee \\ \mathit{NotDistinct}(v, L) & \mathit{AssignInputNotPossible}(v, L)\end{array}}{\mathit{CheckDistinct}(v, L)}$$

The derivation of rules from the static distinctness analysis rules can be understood as follows: the result is distinct if all inputs are distinct (conjunction), and there is no tag conflict (another conjunction). Hence, we apply the conjunction rules from Fig. 5.5: if all inputs to the conjunction can be checked upstream, do so, otherwise perform a local check.

Next, we handle loads with Rules 34, 35, 36 and 37. These rules implement the following logic: if a load either has a base pointer that is not statically or possibly distinct, or else accesses a field on an abstraction that is not statically or possibly distinct, then needed-distinctness must be satisfied with a local check. Otherwise, we can propagate the needed-distinctness upstream to both the base pointer and the field.

Rule 34. *If a load's pointed-to field is not possibly distinct, then it must do a local check if requested.*

$$[\textsc{LoadMustCheckLocally1}]\ \frac{\begin{array}{ll}[x := y.f]_i & \mathit{FieldNotDistinct}(A.f) \\ A \in \mathit{pts}(y) & \mathit{FieldNotPossibleDistinct}(A.f)\end{array}}{\mathit{LoadMustCheckLocally}_1(S_i)}$$

Rule 35. *If a load's pointer is not possibly distinct, then the load must do a local check if requested.*

$$[\textsc{LoadMustCheckLocally2}]\ \frac{\begin{array}{ll}[x := y.f]_i & \mathit{NotDistinct}(y, L) \\ & \mathit{NotPossibleDistinct}(y, L)\end{array}}{\mathit{LoadMustCheckLocally}_2(S_i, L)}$$

Rule 36. *If the result of a field load has a needed-distinctness fact and is not statically distinct, and the load must use a local check by Rule 34 or 35, then perform a local check.*

$$[\textsc{LoadLocalCheck}]\ \frac{\begin{array}{ll}[x := y.f]_i & \\ \mathit{NeedDistinct}(x, L) & \mathit{LoadMustCheckLocally}_1(S_i) \vee \\ \mathit{NotDistinct}(x, L) & \mathit{LoadMustCheckLocally}_2(S_i, L)\end{array}}{\mathit{CheckDistinct}(x, L)}$$

Rule 37. *If the result of a field load has a needed-distinctness fact and is not statically distinct, and the load does not need to use a local check, then propagate the needed-distinctness upstream to both the field (on all abstractions) and the pointer.*

$$[\textsc{LoadPropagate}]\ \frac{\begin{array}{ll}[x := y.f]_i & \\ \mathit{NeedDistinct}(x, L) & \\ \mathit{NotDistinct}(x, L) & \neg \mathit{LoadMustCheckLocally}_1(S_i) \\ A \in \mathit{pts}(y) & \neg \mathit{LoadMustCheckLocally}_2(S_i, L)\end{array}}{\mathit{NeedDistinct}(y) \quad \mathit{NeedFieldDistinct}(A.f)}$$

Finally, we handle stores with Rule 38. There is only one case here, namely propagation from a field's needed-distinctness to needed-distinctness on all values stored into this field w.r.t. loops for which the pointer is not constant. There is no local-check case because we cannot directly check fields for distinctness, as noted above. The possible-distinctness rules will ensure that a field is only possibly-distinct if needed-distinctness can be satisfied by propagation upstream, so there is no need to check for possible-distinctness of the stored values here.

Rule 38. *If a field has a needed-distinctness fact, always propagate a needed-distinctness fact to all stored values passed to store instructions operating on this field without a constant pointer.*

$$[\textsc{StorePropagate}]\ \frac{\begin{array}{ll} [x.f := y]_i & \\ A \in pts(x) & L \in \mathbb{L}(S_i) \\ \mathit{NeedFieldDistinct}(A.f) & \mathit{NotConstant}(x, L) \\ \mathit{NotFieldDistinct}(A.f) & \mathit{NotDistinct}(y, L) \end{array}}{\mathit{NeedDistinct}(y, L)}$$

5.4 Executing with Dynamic Checks

Now that we have determined where to insert dynamic distinctness checks, we need to reason about the runtime aspect of this transform. First, we must actually perform the checks, and propagate distinctness facts dynamically where required. The design of this propagation is key to achieving good performance. Then, we must ensure that we retain soundness in light of the dynamic nature of the check results. If all checks complete successfully, then nothing needs to be done: the parallelization transform was performed based on assumptions that ultimately held, and so execution is sound. However, any failed check suddenly implies a number of derived facts are also now false. We must reason about the lingering effects of such a violation, and introduce fallback mechanisms of some sort to prevent unsound execution. To do so, we introduce a form of synchronization in parallelized loops to avoid the effects of a violated variable distinctness fact.

5.4.1 Variable Distinctness

Performing the Check

The first and most basic type of dynamic check in Icarus is the *variable distinctness* check. The check replaces a statically-proven distinctness fact with a real-time determination, just as a value (for one dynamic instance of the variable-defining statement) is produced, whether that value is indeed distinct.

In order to enable this, we must first modify the definition of variable distinctness slightly.

Recall that variable distinctness was defined in Definition 1 of Chapter 4 as a non-aliasing property: a variable is distinct if it does not alias itself across iterations of a single instance of a loop. This definition is suitable for static analysis such as Daedalus, but suffers one critical flaw as a potentially dynamically-checked property: it refers to the future as well as the past. In other words, if we evaluate the dynamic analogue of static variable distinctness, namely the variable-distinctness of particular values assigned to that variable at runtime, we cannot ever be sure that a variable value is distinct because it *might* alias a value to occur in the future.

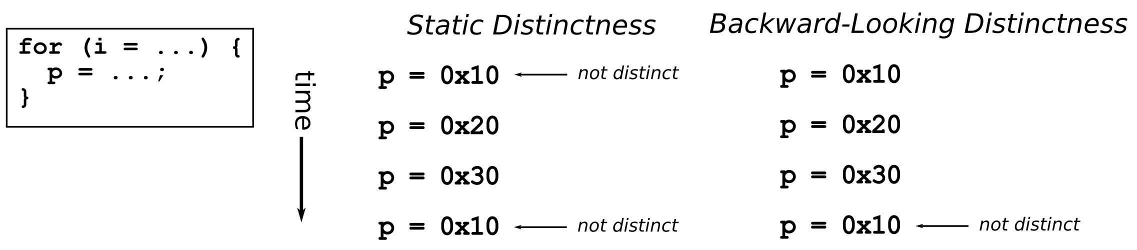


Figure 5.6: The difference between static distinctness, as derived by DAEDALUS, and dynamic backward-looking distinctness, as checked by the dynamic checks in ICARUS.

Fortunately, a simpler definition is possible to dynamically check, and remains sufficient for parallelization. We define *backward-looking distinctness* in Definition 7 and illustrate its difference from static variable distinctness in Fig. 5.6.

Definition 7. *Consider a* particular *iteration i of* one execution *of loop L. Also consider some prior iteration $i' < i$, some value assigned to variable v in iteration i called v_i, and some value assigned to variable v in iteration i' called $v_{i'}$. (A particular variable may have multiple values in one iteration if defined in a nested loop.)*

Distinct(v_i, L) Must not alias a value in a prior iteration: $i' < i \;\rightarrow\; v_i \neq v_{i'}$

Importantly, for our application of loop parallelization, backward-looking variable distinctness on all written-to pointers is *completely sufficient* to show that no cross-iteration dependencies occur. This is for the simple reason that for any pair of aliasing pointer values in different iterations, one must come after the other; hence, the conflict will be detected. (We will see below how this conflict is handled in a sound way.)

In order to determine backward-looking variable distinctness for each particular value, we track the set of values that have occurred so far for this variable in this dynamic instance of the loop (since entry into the first iteration), and in which iteration each value occurred. When a new value is produced, the dynamic-check logic probes the hash table for the value; if found, and if the value was produced in a prior iteration, then this is a violation of the backward-looking distinctness check. Otherwise, we insert the value into the hash table.

Check Synchronization

Note that to return the correct result, the check must have seen *all* prior values w.r.t. the original sequential execution order of the loop. However, if a loop is parallelized, this may not be the same as the order in which values are actually encountered. Consider the loop in Fig. 5.7 and the execution timeline portrayed to its right. Iterations produce the values of `p` out of order, and if the check is simply performed in a given iteration as soon as the value is produced, the check might incorrectly conclude that there is no conflict.

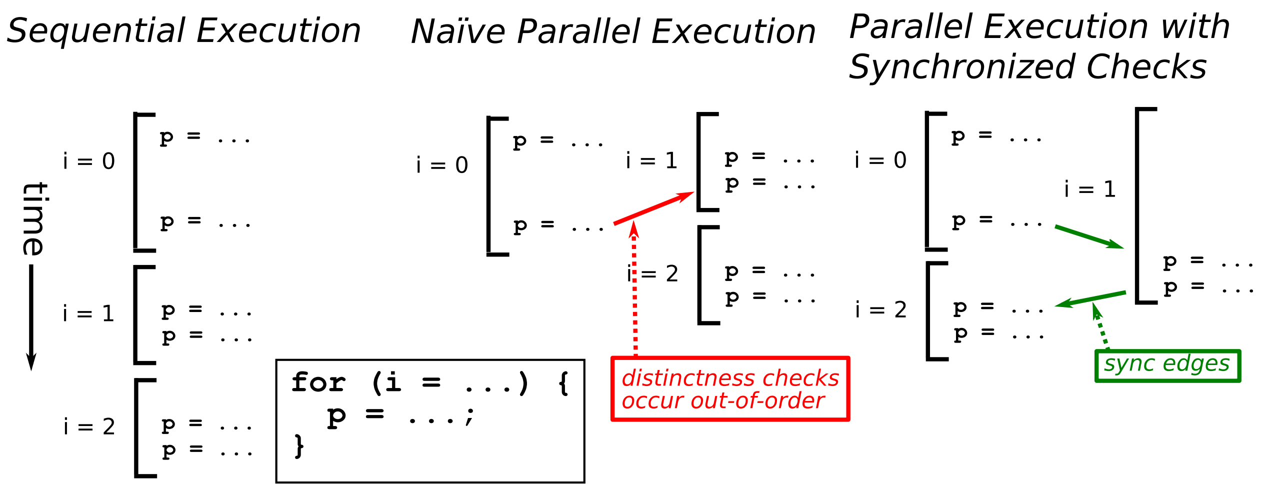
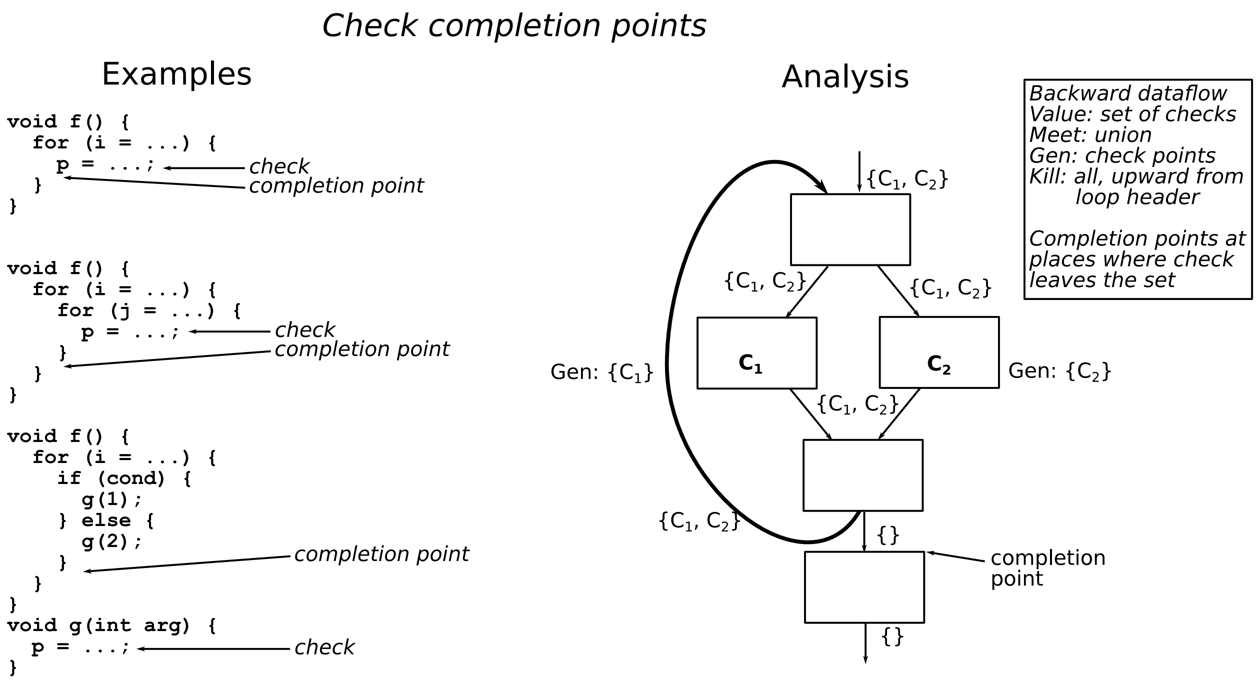


Figure 5.7: Variable distinctness checks require cross-iteration synchronization. The first instance of the dynamic check on `p` in iteration `i = 1` cannot execute until the last instance of the check for `i = 0`, because the check must determine whether `p` aliases *any* value in a previous iteration.

Instead, the check must wait until *all prior values of* `p` have been produced in previous iterations of the loop. (This is analogous to the memory-disambiguation problem in an out-of-order processor's load/store queue: unless the core speculates otherwise and can roll back on a misspeculation, a load must wait for addresses of prior stores to be generated in case they alias.) This synchronization is implemented by finding point(s) in the loop body at which no further instances of a given check occur, which we call the *check-complete points*, and then enforcing a synchronization between the check-complete point in iteration $i-1$ and the first check in iteration i. This is portrayed on the right side of Fig. 5.7.

In order to find these check-complete points, the static analysis that inserts the checks computes a backward-dataflow problem that could be described as a "remaining checks" analysis. The analysis is performed with respect to a particular loop L over its body, intra- and inter-procedurally. The analysis value is the set of checks that might still occur (once or more) before a backedge of L. A check adds itself to the set as it propagates backward. The analysis values meet with the set-union operator at backward merge points. No checks propagate backward from the loop header, because the predecessors to the loop header are either in a previous iteration or not in the loop at all. Finally, once the analysis reaches a fix-point, check-complete points are inserted at the *completion boundary*, where a check leaves the remaining-checks set. Ordinarily, for a simple loop with no nested loops, this will be immediately after the check in question. However, if the check occurs inside a nested loop of L, for example, the check-complete point might be after the exit from the nested loop. This analysis is illustrated in Fig. 5.8.



```
void f() {
  for (i = ...) {
    p = ...;
  }
}
```

```
void f() {
  for (i = ...) {
    for (j = ...) {
      p = ...;
    }
  }
}
```

```
void f() {
  for (i = ...) {
    if (cond) {
      g(1);
    } else {
      g(2);
    }
  }
}
void g(int arg) {
  p = ...;
}
```

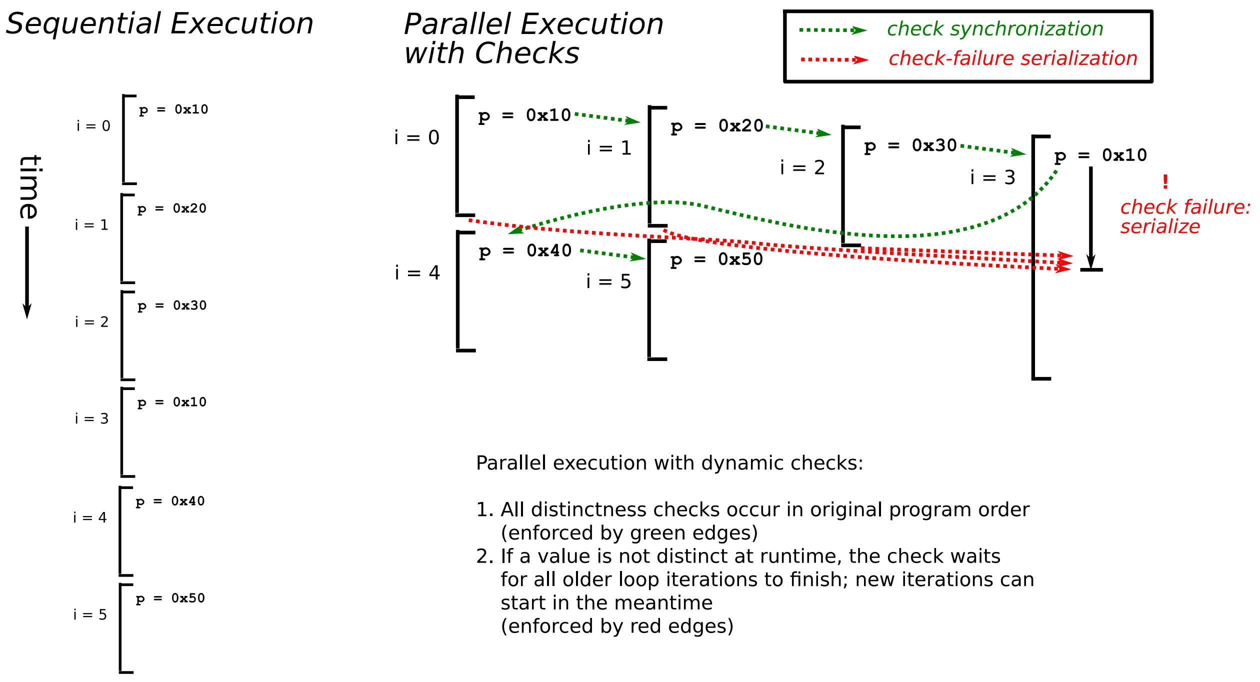
Figure 5.8: Examples of check completion points for (i) a simple check in a single-level loop, (ii) a check in a nested loop, and (iii) a check in a function with multiple call sites. The analysis is a simple backward dataflow analysis illustrated on the right.

Handling Check Failures

Now that we have inserted checks where needed, we must actually make use of their results in order to retain sound execution of the transformed program. As noted earlier, if checks always succeed, then nothing further needs to be done, because the assumptions made during program transformation were unconditionally true (for this execution). However, this is not guaranteed to be the case – otherwise, the checks would not be necessary.

The key insight to our approach is that the execution of a parallelized loop can actually tolerate writes to non-distinct pointers *with the appropriate synchronization*. Specifically, non-distinct pointers only lead to unsound execution if two different iterations with aliasing pointers execute concurrently and interleave their accesses out-of-order relative to the original (sequential) program execution. Thus, if we could synchronize appropriately when a non-distinct pointer is discovered, sound execution would be maintained.

To implement this synchronization, when distinctness check discovers a non-distinct value, it simply waits for all prior iterations to complete (*serializes* on past iterations). This synchronization is illustrated in Fig. 5.9. When the check finally allows execution to proceed, the pointer may be freely used. To see why this is sound, simply consider the loop starting at the serializing iteration to be a *new loop instance*: the



Parallel execution with dynamic checks:

1. All distinctness checks occur in original program order (enforced by green edges)
2. If a value is not distinct at runtime, the check waits for all older loop iterations to finish; new iterations can start in the meantime (enforced by red edges)

Figure 5.9: Parallel execution with dynamic checks: an example of a parallelized loop with one dynamic check failure, showing distinctness-check synchronization at every check instance (green edges) and failing check loop-serialization synchronization (red edges).

partial ordering of instructions is exactly the same as if this were the case. Then the pointer is, virtually, distinct again with respect to the current loop instance.

As a result, when we serialize the loop, we can also clear all distinctness-checking hash tables, because no conflicts with prior values (existing in iterations that have already completed) are possible. This can lead to significantly fewer serializations. Consider, for example, a list with *two* copies of a sequence of objects, i.e., $[x_1, x_2, x_3, \ldots, x_n, x_1, x_2, x_3, \ldots, x_n]$: only one serialization, between the first x_n and the second x_1, is necessary, rather than one for each second occurrence of an object.

Note also that this runtime scheme avoids the need to individually track non-distinct of pointers through program dataflow, because it serializes as soon as it discovers a non-distinct pointer. In effect, the serialization immediately converts the pointer into a distinct pointer (by waiting for the previous copy to disappear) and then continues the program.

There is one unaddressed issue with this approach: although the serialization on previous iterations eliminates any local-variable copies of the non-distinct value, thus rendering the checked value virtually distinct again, it does not erase any *heap effects* of the previous iterations. In particular, consider a loop that stores the checked value in a field on distinct heap objects. We cannot simply proceed with the field store with the assumption that the stored value is distinct, because the multiple copies of the non-distinct value will all be preserved on the heap, and the field will in fact be non-distinct. Seen another way, if we consider the loop to have a new instance after each serialization, then any single store in the original loop will be equivalent to multiple stores, one per virtual instance, which (as we address with the [StoreOverlap] rule in Daedalus) can produce non-distinctness.

In order to properly track such non-distinctness, we actually need to dynamically track the non-distinct state of the particular object field, and handle this appropriately when loading the field value later. We now address this issue in more detail.

5.4.2 Field Distinctness

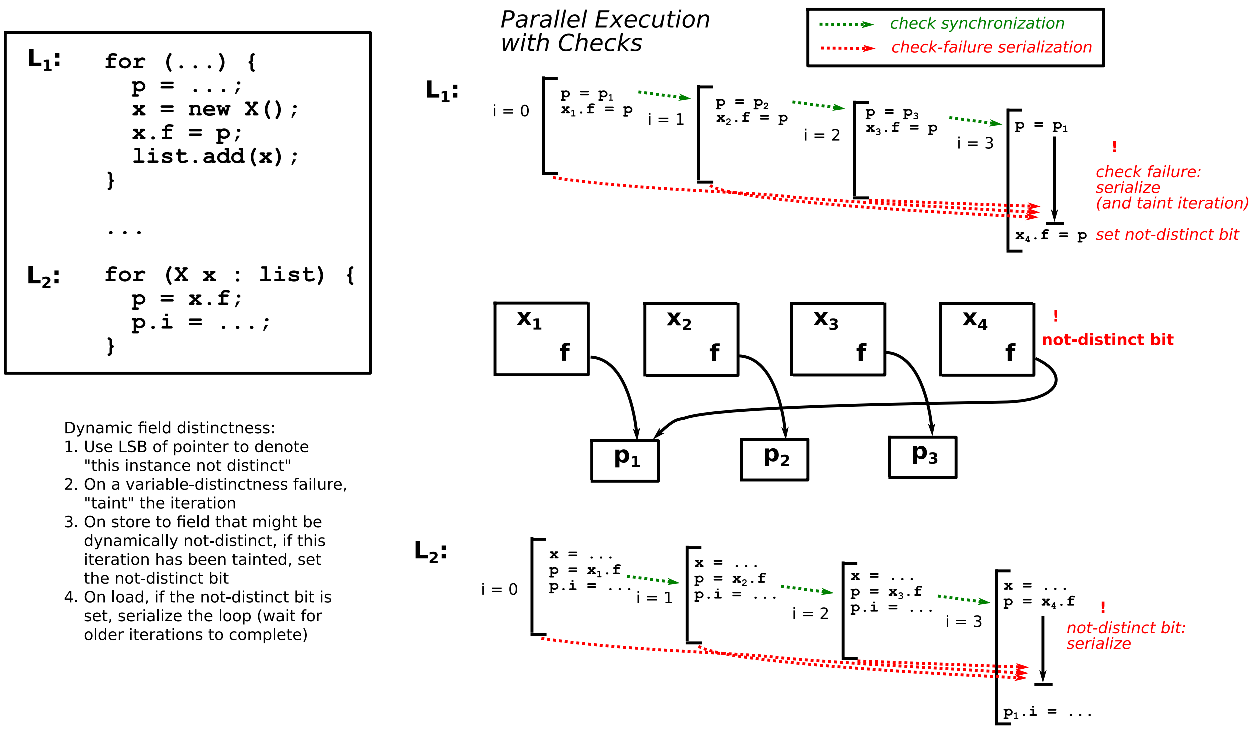
So far, we have described how the runtime system discovers non-distinct variable values and synchronizes on previous iterations of the containing loop in order to maintain soundness. However, our analysis allows *fields* to carry dynamic distinctness facts and be dynamically checked as well. We thus need to (i) detect when a field value is non-distinct, and (ii) perform some fallback action to maintain soundness.

Field distinctness cannot be tested directly, because the check would require enumerating all objects for a heap abstraction at runtime, and the runtime does not keep the metadata necessary to do this (nor would the check be efficient). Rather, field possible-distinctness arises from possible-distinctness of the value stored to the field.

Analogous to dynamic variable distinctness above, we first define a dynamic version of field distinctness. We will take a *unidirectional* approach here, just as we took a *backward-looking* approach for variables, because of the insight that an aliasing pair of fields only needs to be detected on one side of the conflict. In particular: for two different objects x_1 and x_2 represented by the same heap abstraction X with dynamically-distinct field $X.f$, whenever $x_1.f = x_2.f$, *either* $x_1.f$ is dynamically field-non-distinct *or* $x_2.f$ is.

This dynamic, field-specific notion of distinctness must be tracked at runtime, and we do so as follows (illustrated in Fig. 5.10). First, we keep a 1-bit flag alongside the field to track its dynamic (non-)distinctness. Because the field is a pointer, and pointers must be aligned to some alignment > 1 (on typical architectures and runtime platforms, including the JVM that we target), we employ the common "tagged pointer" trick and store this not-distinct flag in the lowest bit of the pointer.

The most precise implementation of a field-distinctness check would carry a *non-distinct taint* from a variable that is determined to be non-distinct, and then add this taint to any field to which the value is stored. However, this would likely impose too much runtime overhead, and is an invasive change to the program. Instead, we keep a single "tainted" flag for the current iteration of a given loop. This flag is set by variable dynamic-distinctness checks. Importantly, the check for the taint-bit is slightly different: it probes a different hash-table that is *not* cleared on loop serialization, instead growing for the duration of the loop



```
L1:   for (...) {
        p = ...;
        x = new X();
        x.f = p;
        list.add(x);
      }

      ...

L2:   for (X x : list) {
        p = x.f;
        p.i = ...;
      }
```

Dynamic field distinctness:
1. Use LSB of pointer to denote "this instance not distinct"
2. On a variable-distinctness failure, "taint" the iteration
3. On store to field that might be dynamically not-distinct, if this iteration has been tainted, set the not-distinct bit
4. On load, if the not-distinct bit is set, serialize the loop (wait for older iterations to complete)

Figure 5.10: Illustration of dynamic field distinctness propagation with the use of "not-distinct" bits on pointer fields.

instance. Or, equivalently, a single hash table records the iteration number that last produced a value, and this is compared against the last serialization point. (The intuition here is that although the serialization flushes the iterations with conflicting pointers out of the system, it does not flush their side-effects on the heap, so we must remember the pointer-set for the duration of the loop instance.) Then, on a store to a field that is dynamically distinctness-checked, if the tainted-iteration flag for any loop in context is set, the not-distinct bit is set on the stored pointer value.

Finally, when a pointer is loaded from a dynamically-checked object field, the load checks the not-distinct bit before masking it off the loaded value. If the bit is set, then the transformed program immediately serializes the loop, waiting for all older loop iterations to complete. In addition, all newer iterations must wait for the current iteration to complete. (The reasons for this become clear in our soundness argument in Appendix B.)

5.4.3 Must-Alias Checks

One final type of check is necessary for soundness. Recall that the loop parallelization rules of Daedalus require not only that written-to pointers are distinct, but that pointers that may alias, i.e., write to the same heap abstraction, *must* alias (§4.4). Otherwise, two distinct pointers might traverse the same distinct objects in different orders, thus aliasing across iterations.

Rather than disqualify such loops, Icarus simply inserts additional checks. The fundamental condition to verify is not that the pointers must alias, but rather that among *all pointers that access the same may-alias heap abstraction*, there is distinctness. In other words, no two of these possibly-aliasing pointers can have the same value across iterations.

In particular, Icarus inserts variable-distinctness checks on all such pointers when multiple written-to pointers in the loop may alias. However, unlike above, these checks all have the same check identifier, hence check against the same duplicate-detection hash-table. Additionally, the check-completion point is calculated with respect to all of these checks.

5.5 Evaluation

In order to evaluate the effectiveness of a hybrid dynamic-static system, we must answer two fundamental questions. First, do the dynamic checks reveal additional opportunity for optimizations, such as loop parallelization? Second, what is the overhead of performing the dynamic checks at runtime?

We answer these questions with several experiments using an implementation of Icarus based on the Daedalus infrastructure. We will first simply report the number of additional parallelizable loops (counting statically) and the number of inserted dynamic checks required to parallelize these loops. Then, we measure improvement in *coverage*: the fraction of all dynamic instructions in a trace that are within the body of a parallelizable loop. We finally show execution speedups under several configurations.

This evaluation is designed mainly as an opportunity study: because our primary contribution is the analysis itself, the parallelization opportunity that it uncovers is a more fundamental measure of its effectiveness than the simulated speedup of a particular configuration. In addition, as we will discuss further below, we find that further work is required to tune the heuristics (e.g., loop selection) and perhaps optimize the dynamic checks further. We wish to decouple the potential that the analysis itself provides, and more generally that the hybrid static-dynamic approach provides, from our particular implementation.

5.5.1 Methodology

We implement Icarus on top of the Daedalus system in several parts. We first implement the analysis itself. As before, the analysis is constructed as a set of inference rules in Datalog. The result of the analysis is a set of annotations on program points indicating parallelizable loops and locking-insertion directives (as in Daedalus) as well as dynamic check sites.

We then evaluate the system in simulation. We first propagate program values through the dynamic instruction traces so that the simulation infrastructure can actually evaluate the dynamic checks. This requires emitting additional metadata in the traces and tracking the current state of JVM local variables and operand-stack slots. The dynamic checks are actually evaluated in the trace demultiplexer, rather than in the simulator itself, because all of the needed values are present in the trace, and the check results will not change because loop parallelization retains original program semantics. As the demultiplexer evaluates dynamic checks, it emits the appropriate synchronization edges between program points in the trace chunks in order to enforce the necessary loop serializations and dynamic check ordering.

We initially performed system simulations with the same configuration as that used for DAEDALUS (see Table 4.1). As we will describe in more detail in §5.5.5, we found that the backend loop-choice heuristics and dynamic-check implementation are not yet well-suited to the large number of loop iterations that are dynamically parallelized (and often serialized). We thus perform another set of simulations as an ideal limit study, showing potential with workqueue iteration-spawn latency and cross-core cache-miss latency removed.

Note that the coverage and simulation speedup results for a given benchmark in this chapter are not directly comparable to those for DAEDALUS in Chapter 4 because we needed to re-trace benchmarks in order to capture program values and other additional metadata. Because capturing these values enlarged traces significantly, the traces have somewhat lower instruction counts, and so spend less time in the main loops of benchmarks; coverage is thus slightly lower than it would be in a real system evaluation with longer runs. In order to allow direct comparison with DAEDALUS, we re-run all DAEDALUS evaluations here using the same traces and configurations as for ICARUS. The benchmarks themselves are exactly the same, so the static analysis results (e.g., number of parallelizable loops) are directly comparable.

5.5.2 Parallelizable Loops

We first evaluate the simplest metric of success: how many additional loops can be parallelized with the help of dynamic checks? Table 5.1 shows, for each benchmark, the number of loops that are parallelizable under the rules of the baseline affine-indexing analysis, DAEDALUS, and ICARUS. In addition, for ICARUS, the table shows the number of each type of dynamic check required: variable-distinctness checks, tag-mismatch checks, load not-distinct-bit checks, and store not-distinct-bit updates. As shown by this data, ICARUS is able to discover a significant number of additional loops, and the number of required checks is modest. If a fully-dynamic scheme were used that checked every address for a conflict (e.g., TLS), many more checks would be required. Instead, ICARUS is able to statically propagate the implications of a few checks and use these facts in many places.

	Parallelizable Loops		Dynamic Checks			
Benchmark	Stat.	Dyn.	Var	Tag	Load	Store
dacapo.batik	62	68	3	196	0	0
dacapo.luindex	0	0	0	0	0	0
dacapo.pmd	36	46	22	14	2	38
dacapo.xalan	0	1	0	6	0	0
cpu.cloudsim	15	20	98	24	8	7
cpu.djbdd	9	21	29	34	0	0
cpu.jacc	42	62	43	18	0	0
cpu.jgrapht	5	5	0	0	0	0
cpu.jlatexmath	0	0	0	0	0	0
cpu.jscheme	0	0	0	0	0	0
cpu.jtidy	5	9	533	78	545	73
cpu.sablebdd	10	11	1	0	0	0
cpu.sat4j	9	11	2	6	0	0
olden.bh	10	10	0	0	0	0
olden.bisort	0	0	0	0	0	0
olden.em3d	1	2	6	4	1	6
olden.health	0	0	0	0	0	0
olden.mst	2	2	0	0	0	0
olden.perimeter	0	0	0	0	0	0
olden.power	12	14	1	45	0	0
olden.treeadd	0	0	0	0	0	0
olden.tsp	0	0	0	0	0	0
olden.voronoi	0	0	0	0	0	0
pbbs.comparisonsort	0	1	0	6	0	0
pbbs.convexhull	0	0	0	0	0	0
pbbs.integersort	0	3	2	6	0	0
pbbs.nn	2	5	13	0	0	0
pbbs.raycast	3	6	1	29	0	0
pbbs.removeduplicates	0	0	0	0	0	0
Average	7.7	10.2	26.0	16.1	19.2	4.3

Table 5.1: Parallelizable loop and dynamic-check counts under ICARUS compared to DAEDALUS.

5.5.3 Dynamic-Check and Loop Parallelization Success Rates

Next, we count the number of dynamic checks, and the number of dynamically-checked parallelized loop iterations, in each benchmark. We report the fraction of each that are successful: checks that verify the dynamically-checked fact (hence require no corrective action), and loop iterations that do not experience any check failures (hence do not need to serialize on previous iterations). Table 5.2 presents this data: for each benchmark with at least one executed dynamically-parallelized loop, it provides the number of successful and failed checks, the number of successful and serialized dynamically-parallelized loop iterations, and success ratios for each.

Overall, check success rates and the resulting loop-iteration success rates vary widely by benchmark. The analysis has no heuristics for which distinctness checks might be likely to succeed or fail, other than the

	Dynamic Checks			Loop Iterations		
Benchmark	All	Success	Succ. Rate	All	Success	Succ. Rate
dacapo.pmd	728	651	89.4%	40	29	72.5%
cpu.cloudsim	1241	684	55.1%	339310	339151	99.9%
cpu.djbdd	223	200	89.7%	227	204	89.9%
cpu.jacc	749	661	88.3%	1008	943	93.6%
cpu.jtidy	2342	2071	88.4%	2348	2077	88.5%
olden.em3d	35000	26336	75.2%	5200	344	6.6%
olden.power	56008	54999	98.2%	78692	39683	50.4%
pbbs.comparisonsort	0	0	—	10002	10002	100.0%
pbbs.intsort	200090	100060	50.0%	110132	10102	9.2%
pbbs.nn	1017	777	76.4%	82669	82429	99.7%
pbbs.raycast	81088	81038	99.9%	2510	2491	99.2%

Table 5.2: Success rates for dynamic checks (left half) and dynamically-parallelized loop iterations (right half) for benchmarks that have at least one dynamically-parallelized loop under Icarus.

very simple rule that a variable proven to be constant with respect to L will never be distinct with respect to L. Thus, it is not surprising that by maximizing the number of parallelizable loops, Icarus has also chosen to perform a number of checks that are unlikely to succeed. Nevertheless, the fact that in a number of cases, a large fraction of loop iterations succeed (execute in parallel without serializing), is encouraging. Our observations here – simply that many actually-parallelizable iterations exist when observed dynamically – are in line with previous observations by speculative-parallelization works. (The key difference between Icarus and those systems, aside from fewer required checks, is that our system is able to parallelize these loops without any speculation.)

One additional interesting observation is that in several benchmarks (`pbbs.comparisonsort`, `pbbs.nn`, and `pbbs.raycast`), no dynamic check ever actually executes, despite the dynamic parallelization of at least one executed loop. This means simply that the path on which the check was placed was never reached. (The runtime still has slight overhead in this case: when execution reaches a point in the loop iteration that can no longer reach any check in that iteration, the check-completion point will execute to allow future iterations to check against the distinctness hash-table.) This is the best case for a dynamic check: it is inserted for soundness, but is not actually reached under real inputs, so has as little overhead as possible.

5.5.4 Parallelization Coverage

We next evaluate parallelization *coverage*, using the same definition as in §4.6.2: fraction of dynamic instructions in an execution trace that are within an iteration of a parallelized loop. Fig. 5.11 shows this data for all benchmarks with nonzero coverage. We break down coverage for each benchmark into several portions: instructions in loop iterations that are statically parallelizable under the affine-indexing array-based baseline analysis, then instructions in parallelizable loops under Daedalus, then those in iterations

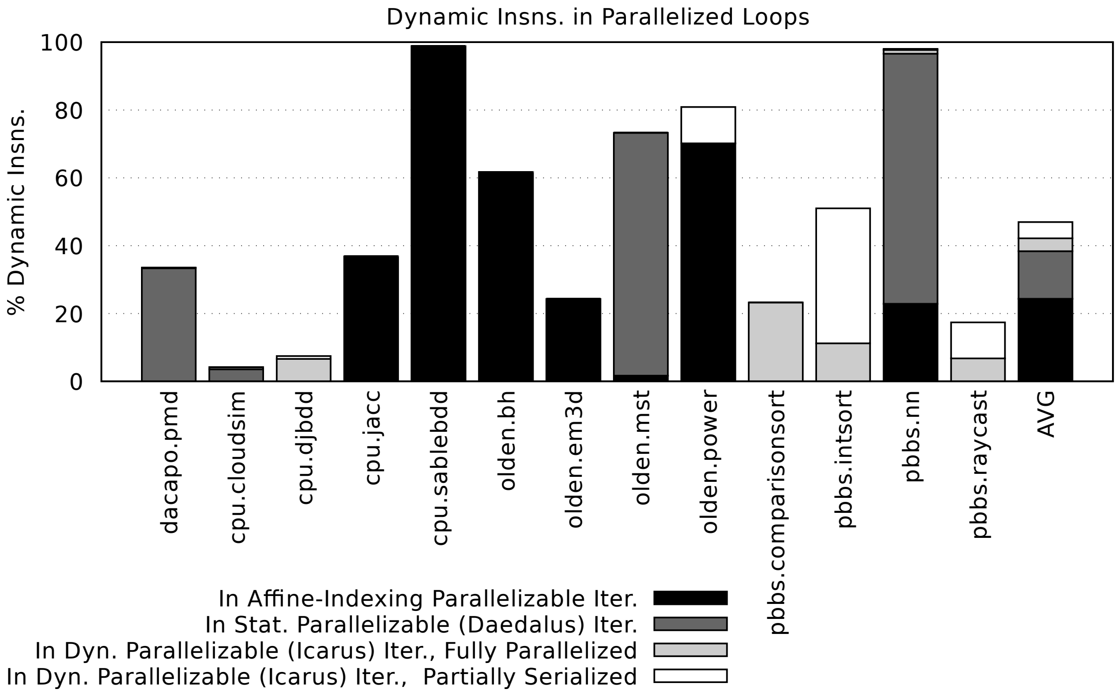


Figure 5.11: Parallelization *coverage*, or fraction of all dynamic instructions in the evaluated trace that occur within a parallelizable loop instance, for benchmarks with nonzero coverage under any analysis. This plot breaks down coverage by (bottom to top) parallelizable loops under the baseline affine-indexing/array-based system, then statically-parallelizable loops under DAEDALUS, then iterations of dynamically-parallelizable loops under ICARUS that did not serialize due to a failed check, and finally iterations of dynamically-parallelizable loops under ICARUS that did serialize.

that are dynamically paralellizable under ICARUS and had no failed checks, and those in iterations that are dynamically parallelizable but had at least one failed check. Note that the last category may still expose parallelism during execution because the serialization only occurs at the point that the non-distinct value is actually generated. For example, if a non-distinct pointer value is computed and then accessed near the end of a loop iteration, most of the iteration may still execute concurrently with other iterations.

As above, the effects vary widely by benchmark, but in a number of cases, ICARUS is able to substantially increase coverage. Overall, coverage on this set of benchmarks increases from 24.3% in the baseline affine indexing system and 38.4% using DAEDALUS to 42.2% using ICARUS, counting only iterations with all-successful dynamic checks, up to an upper bound of 47.0% in serialized (but still potentially partially-parallel) iterations.

5.5.5 Execution Speedup

We next simulate execution of the parallelized loops discovered by ICARUS to measure program speedup. We first evaluated ICARUS on the same system configuration that was used to evaluate DAEDALUS, and quickly found that speedup was not optimal: on average (geomean), ICARUS was actually 0.3% *slower* than

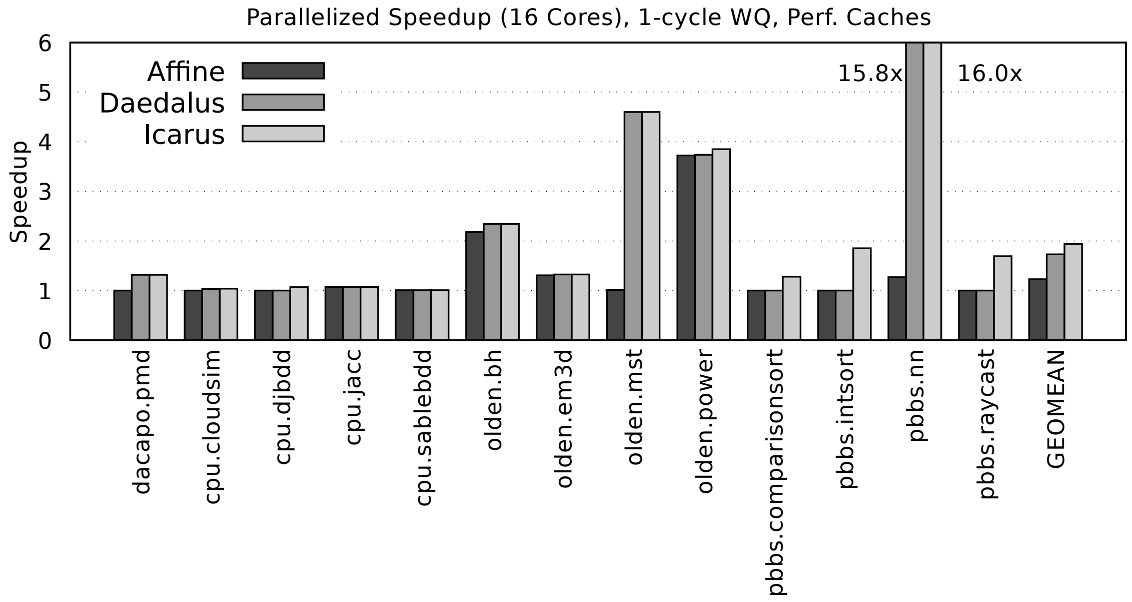


Figure 5.12: Parallelization speedup for Icarus as compared to Daedalus and the affine-indexing system on an *ideal* configuration removing the overhead of suboptimal loop-choice heuristics.

Daedalus, with significant losses in some benchmarks.

The immediate conclusion from this simulation result is that despite the promise shown by lower-level metrics such as successful dynamic checks and dynamically-parallelizable loop iterations, Icarus has little effect over Daedalus. However, given that many loop iterations are successfully parallelized, and did not require any serializations to maintain soundness, we expect that there should be *some* non-negligible speedup. In particular, in an idealized analysis, a loop that is parallelized with dynamic checks should at worst perform the same as the original sequential loop: serializations simply force one iteration to begin or resume after the previous iterations complete. However, we found that this expected speedup is largely masked by the overhead of two factors: *workqueue latency* and *cross-core cache misses*. Both of these aspects of the system become relevant because Icarus (i) finds many small loop iterations, magnifying the cost of the necessary inter-chunk synchronization, and (ii) parallelizes many more loop instances in general, which allows execution to migrate arbitrarily across cores. (Our system does not have a core affinity policy – ready chunks are assigned to the first available core.)

To measure the *potential*, given sufficient backend engineering, we ran one more round of simulations. Fig. 5.12 shows an evaluation of nonzero-coverage benchmarks with two changes: (i) the workqueue latency is reduced to 1 cycle, which eliminates the overhead of the additional synchronization from many small iterations; and (ii) the cache system is disabled completely, so that all memory accesses are cache hits, thus eliminating the cache penalty when work migrates across cores. This evaluation shows somewhat more

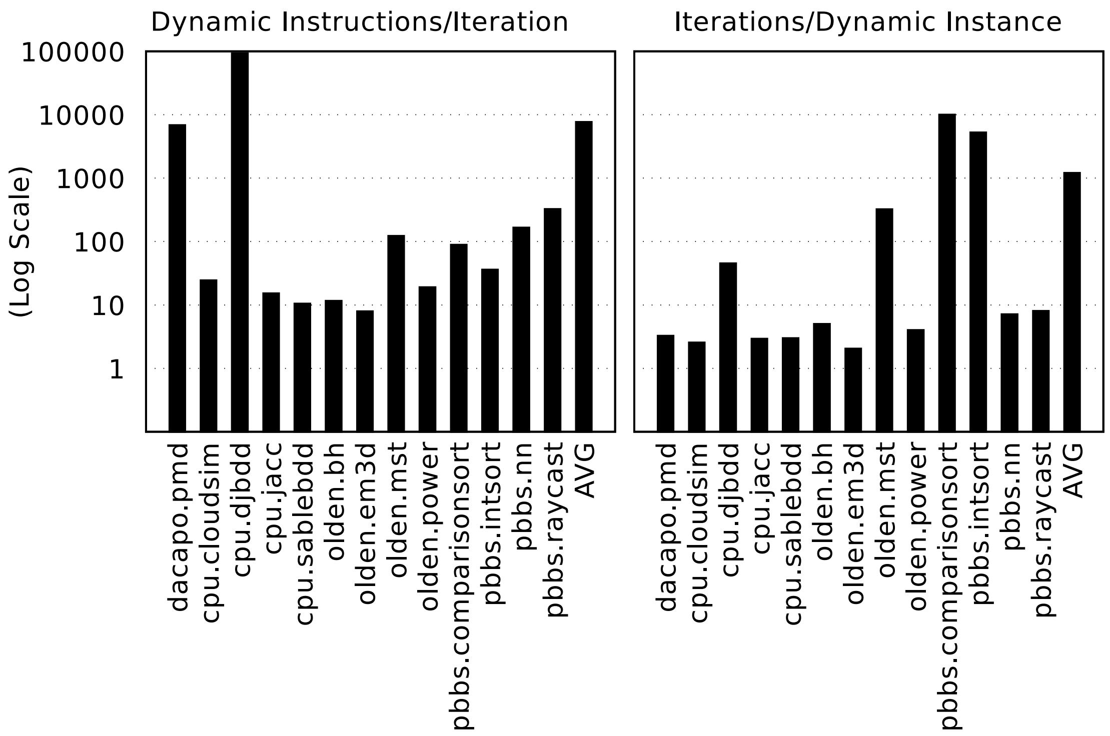


Figure 5.13: Average loop iteration length (in dynamic instructions) and iteration count per dynamic instance for all parallelized loops under Icarus. These results quantify the remaining difficulty in obtaining good speedup with real-world runtime parameters.

promise, at least on a few benchmarks: on average (geomean), Icarus achieves a geomean speedup of 1.94× on 16 cores, relative to 1.73× for Daedalus and 1.23× for the affine-indexing-based system.[2] With more work, the potential could likely be extended further. Though this is strictly an upper bound, with important overheads eliminated, it demonstrates that a sufficiently well-engineered parallel loop runtime could likely achieve speedup by using dynamic checks. We therefore suggest that Icarus, as an analysis, points toward and enables further work in program parallelization by soundly proving many loop iterations to be parallelizable.

5.5.6 Loop Iteration Length and Count

Finally, we present data that quantifies the average length (in dynamic instructions) of parallelized loop iterations, and the average number of loop iterations per dynamic loop instance, in the programs parallelized by Icarus. Fig. 5.13 shows these results.

Overall, these measurements demonstrate the main challenge to *practical* loop parallelization with a system such as Icarus: though programs contain quite a few parallelizable loops on average, these loops

[2]Note that these evaluations do not profile for optimal loop choices in the affine-indexing and Daedalus systems as the evaluation in Chapter 4 did, and even on this nearly-ideal system there are still some overheads for parallel loop synchronization, so the results are not directly comparable to the previous chapter.

tend to be dynamically quite small. Iterations tend to be short, and iteration batching cannot amortize per-iteration overhead effectively because there are also relatively few iterations in many loop instances. Thus, both coordination/scheduling work per iteration and per dynamic instance must be lightweight in order for speedup to result.

5.6 Discussion

5.6.1 Practical Application to Loop Parallelization

We have presented Icarus with a focus on the analysis itself, simply demonstrating that a sound means exists to extend a static analysis with a few well-placed dynamic checks. Our proof-of-concept implementation of the execution mechanisms, though able to expose opportunity in available parallelism, highlighted the need for more backend engineering to obtain better real-world speedups. In this section, we discuss a few directions for future work along this line.

Perhaps the largest payoff exists in correctly choosing which loops to parallelize. We have adopted the simplest possible heuristic: we parallelize all loops that can be parallelized. As a proof-of-concept, this exposes maximal potential parallelism. It works well to provide an upper bound, as we have done in our evaluation (§5.5.5). However, many static and dynamic techniques could be used to improve this choice. For example, the system could statically profile the parallelization of each loop to determine which is actually beneficial. A more sophisticated dynamic adjustment could be made using traditional feedback-controller techniques or machine learning. More detailed cost models and other heuristics might improve the choice at compile time. Such a choice could even be made dynamically: any parallelizable loop can compute its trip-count before it begins executing (this is among the parallelizability requirements), so the program could branch to a parallelized version of the loop or the original code separately for each instance. Though this work is orthogonal to our analysis-focused exploration, it is likely necessary for Icarus to be beneficial in a real compiler/runtime implementation.

It might also be possible to harness the opportunity in smaller loop instances with the assistance of a very high-performance parallel loop runtime, or perhaps some hardware for managing the loop iteration workqueue. While such an approach is interesting, it extends beyond the scope of this thesis, and we leave it for future work.

We observe that there is a *continuum* between Icarus, which is fully non-speculative, and speculative parallelism systems such as TLS (Thread-Level Speculation). We believe that there is significant opportunity to be found by exploring hybrids of the two. Icarus is carefully designed to check every dynamic fact before executing code that relies on it; if it instead performed some judiciously-applied speculation, limited in scope by the static analysis, it could avoid the check-ordering synchronization in many or most cases. The speculate/block choice could be made individually for each memory access or pointer variable based on an estimate of the likelihood of the check to succeed. (Note that this is analogous to the decision to speculate or block a store-to-load forward in an out-of-order CPU microarchitecture, and there are predictors to make this decision as well [32].)

There are also program transforms that could be applied on top of loop parallelization to reduce the impact of the synchronization. Most notably, Zhai et al. [116] propose altering a compiler's instruction scheduler to take TLS loop iteration critical-path length into account. A similar transform could ensure that, in effect, all or most potentially non-distinct addresses are generated early in the loop iteration so that younger iterations can quickly check and continue.

5.6.2 Limits of Sequential-Program Auto-Parallelization

Finally, we briefly discuss our interpretation of the implications of our results on the broader problem of auto-parallelization. While we saw significant parallelization potential in many programs, the opportunity was far from universal. In many other programs, important loops remain serialized because of true dependencies that are either fundamental to the particular algorithm, or else are avoidable but only with truly deep changes to the program.

For an example of an incidental dependency that *could* be removed, consider the benchmark `cpu.janino`, which is a Java compiler. In this program, a loop processes input files one at a time for compilation. As each file is compiled, the name-resolution logic looks up and processes referred-to classes (e.g., in the standard library or in other portions of the program not currently being compiled). The first time a class is referenced, it is processed in certain ways and inserted into a symbol table. Subsequently, this cached data is simply fetched from the symbol table. This pattern is fundamentally parallelizable, but with some complexity: the program could perform the processing work in whichever loop iteration first finds that the results are not cached. Then, to avoid redundant work, the thread that computes the result would likely insert a *placeholder value* of some sort to indicate that other threads should block until the result is computed. Such an analysis and transform is currently beyond the scope of our system's capabilities.

Many programs, unfortunately, have examples of *fundamental* dependencies: many algorithms are inherently serial. For some of these algorithms, alternative algorithms exist that solve the same problem in a parallel way. However, replacing the former with the latter is a pattern-matching problem that is unlikely to be easily generalized.

Overall, engineering the parallelizability analysis and set of transforms is a tradeoff between general applicability and success in specific applications. At one extreme, the system pattern-matches all algorithms or main loops that we are interested in parallelizing, and replaces them with their parallel equivalents that have been hand-selected by the tool author. At the other extreme, the system analyzes all program code at the language level and performs only transforms that retain the same computed result with respect to language-level semantics. While no system can solve all parallelization problems (due to the Halting Problem), there several useful design points along this spectrum. Daedalus and Icarus lie closer to the latter end of this spectrum, i.e., general analysis, and fundamentally improve on the state of the art by enhancing the heap analysis, allowing the compiler to prove that more parallelizations will preserve original behavior. By adding better recognition of particular patterns and idioms that have parallel, or parallelization-friendly, equivalents, the system could be made to parallelize more code.

5.7 Related Work

As far as we are aware, no prior work provides a principled means of converting a static analysis to a hybrid static-dynamic analysis as we have. However, previous work has suggested several ways to use dynamic checks to enhance or replace static analysis.

As cited earlier, Wu and Padua [114] introduce a loop-parallelization system that uses dynamic checks on hash-table keys to verify that the keys do not overlap between loop iterations. It appears that this design choice was motivated out of necessity, however, rather than as a means to improve precision. Nevertheless, the system demonstrates a hybrid between static and dynamic analysis.

Ernst [41] discusses the tradeoff between static and dynamic analysis. This prior work notes that static analysis is sound but often limited in precision; in contrast, dynamic analysis is perfectly precise, but unsound. Ernst suggests harnessing the best of both approaches by creating hybrid analyses.

Several analyses make practical use of this insight. For example, Sengupta et al. [98] build a system that enforces a strong memory model with a combination of a static compiler analysis and dynamic instrumentation. In general, compiler analyses combined with particular runtime techniques can work together to provide useful guarantees to the programmer: for example, a series of works on intermittent-computing programming models [70, 34, 71, 72] use a combination of static compiler transforms and transaction-like runtime system support to provide coherent program state on a system that fails and restarts frequently. Rus et al. [94] introduce a hybrid static-dynamic analysis framework that can incorporate multiple loop-parallelization analyses, though it is specialized for numeric/scientific-type loop nests with array accesses.

Specialization is another compiler optimization that involves both static and dynamic elements. A specialization transform takes a portion of the program and hypothesizes certain assumptions, such as the types of variables or the aliasing or non-aliasing of pointers. Then, given these assumptions, it can often perform much more powerful optimizations because edge-case scenarios are no longer possible. For example, many high-performance JIT compilers for dynamic languages, such as Chromium's V8 [8] and Firefox's SpiderMonkey [7], specialize on variable types. Chevalier-Boisvert and Feeley [31] introduce Basic Block Versioning, in which assumptions about variable types are checked once dynamically (e.g., by a branch) then propagated statically along control-flow edges from this branch, similarly to how Icarus statically propagates checked assumptions from an earlier dynamic distinctness check to later direct and indirect uses of the value.

Motivated by conservative static analysis results for loop parallelization, many purely-dynamic dependency tracking techniques (e.g., profiling read and write sets) have also been used in an effort to measure loop dependencies more precisely [17, 79, 90, 60, 52, 96, 106]. The main disadvantage of such systems is that purely-dynamic analyses cannot be sound: some new program behavior could always occur that did not occur during the analysis phase. In addition, these analyses typically have high overhead during profiling because they must instrument every memory access. In contrast, Icarus uses static analysis to make best use of a dynamically-checked fact by following all of its direct and indirect implications.

Finally, our approach to executing parallelized loop iterations resembles Thread-Level Speculation [104] and other speculative parallelization systems such as Multiscalar [101], though only superficially. The main similarity between these approaches and ours is in the use of dynamic checks across concurrently-executing loop iterations to ensure that memory accesses do not conflict. The primary difference is that these approaches are speculative, resolving conflicts by rolling back state, while our approach avoids speculation by pausing loop iterations until no conflict exists.

5.8 Chapter Summary

In this chapter, we introduced Icarus, a system that extends the static loop-parallelization analysis of the Daedalus system to allow for *dynamic* checks of certain required program properties. The hybrid static/dynamic approach used by Icarus determines which dynamic checks are necessary to prove that a loop is parallelizable, propagating the analysis conclusions that are conditionally true (given a successful check) and thus allowing a single dynamic check to unlock many additional static-analysis facts and resulting program transforms. We described how to transform inference rules from Daedalus to produce the two-pass analysis of Icarus that computes (i) possible dynamically-verified facts, and (ii) needed checks. We outlined a runtime system that performs dynamic variable distinctness checks, and propagates not-distinct flags on object fields. This system inserts synchronization between loop iterations as necessary to preserve sound execution without any speculation or rollback of side-effects. Finally, we measured the parallelization coverage of this analysis and its effectiveness in simulation, showing that there is significant potential to improve performance if the backend execution challenges are overcome.

Chapter 6

Future Work and Conclusions

There once was a quite large compiler
That with many tools did conspire.
It became soon aware
That it could not stand to bear
More code, and promptly retired!

6.1 Future Research Directions

In this thesis, we began with a vision of *high-level program transforms* by a compiler that understands the programmer's intent. We have introduced several analyses that provide a technical underpinning to enable a small step toward this goal, specifically by enabling better loop parallelization. However, there are many more steps that a compiler could take to recognize and take advantage of common patterns and idioms in programs. We outline several directions for future work here.

6.1.1 IR with Additional Primitives

Our built-in data structure primitives introduced in Chapter 3 enabled increased precision in points-to analysis, and thus also distinctness analysis, by encoding program operations on some fundamental data structures explicitly rather than as a black-box implementation. However, there are two shortcomings to this approach that could be addressed with future work.

First, the set of primitives (maps and lists), while very useful and general, does not cover the full range of interesting data structures in use today. The user may build (e.g.) a graph out of lower-level data structures such as lists, but by analyzing the code only at the level of the constituent lists, the compiler is missing the bigger picture: for example, it would not be able to recognize a BFS traversal over the graph.

Thus, we suggest incorporating additional built-in operations, but in a principled way. In particular, we envision a *hierarchical* IR, where a set of operations could be represented by a new, higher-level operation, but the original operations would remain as well. In essence, this involves (i) an open-ended definition of IR statement types, and (ii) a combinator that joins *alternative but equivalent* implementations. (This is somewhat similar to PetaBricks [14], though the alternative views are provided *to* the compiler rather than chosen *by* the compiler, and the analysis tends to *raise* the level of abstraction rather than *lower* it to a concrete implementation.)

In addition to the existing built-in data-structure operators, which are fundamentally data-manipulation statements, we suggest adding higher-level *control-flow* primitives as well. For example, a traversal loop over an iterable sequence, which we recognized manually in DAEDALUS for the purposes of loop parallelization, could be its own primitive. Its fallback representation is the ordinary loop using the iterator-related operators, but it would serve many analyses well to see the higher-level pattern, too.

The advantage of such a representation is that it would enable additional high-level understanding to be incorporated into the compiler's model of the program, while retaining the original operations, so that analyses need not be extended to understand every new high-level primitive. In other words, if an analysis directly understands (e.g.) a BFS graph-traversal control-flow operator, it can use this view, but otherwise, an analysis that understands only the lower-level list and map operations will work too.

Once this hierarchical, extensible representation exists, there are multiple ways to lift a program into the higher-level representation. The extended primitives could be used directly by a library that provides both the direct high-level operation and a fallback lower-level equivalent implementation. Or, the creator of the new IR primitives could develop *pattern matchers* to lift the program into this representation automatically. We now discuss this idea in more detail.

6.1.2 Higher-Level Pattern Recognition and Transforms

A compiler analysis generally falls into one of two categories: it could be a general analysis that applies to all code (e.g., constant propagation, common-subexpression elimination, points-to analysis, as well as our distinctness analysis), or it could consist of a set of pattern-matching rules that transform certain idioms into better implementations (e.g., strength reduction and other peephole optimizations). This thesis has focused on introducing general analyses that provide useful information to enable desired program transforms. In particular, semantic models, distinctness analysis and dynamic checks refine the aliasing information that is available to improve loop parallelization. In addition to this work, many different types of pattern-matching could be employed.

We suggest considering the use of pattern-matching to find (at least) (i) functional-programming idioms (such as maps, filters, and reduces over sequences) that have been written imperatively, and to encode them directly as data-structure operators in a sort of data-streaming DSL representation; and (ii) well-known sorts of traversals over data structures, such as DFS and BFS over graphs, and encode them as control-flow operators. These patterns could naturally compose as well: for example, a functional map pattern-matcher could be expressed in terms of an iterable-sequence loop pattern. The map pattern also requires a loop parallelization-like analysis to ensure each item's computation is independent, so it could match on results provided by distinctness and parallelizability analysis, as could many other patterns. The "output by appending to a list" portion of the imperative map-over-list implementation could be factored out and reused by (e.g.) the filter pattern as well.

The ultimate goal of this pattern matching is to encapsulate as much of the program as possible in higher-level descriptions to enable more powerful transforms to be done. Remaining behavior that cannot be analyzed either becomes a black box inside a higher-level control-flow operator (e.g., a traversal loop that is marked as *not* parallelizable because of arbitrary data dependencies), or else arbitrary glue code at the top level, between islands of high-level understanding.

6.1.3 Automated Transform of Analyses to Use Dynamic Checks

We believe that many other program analyses could benefit from the use of dynamic information. To that end, the techniques that we developed for Icarus are reasonably general, and could likely be applied to many other analyses. The principles to create the inference rules for the two-pass dynamic-fact possibility/need analysis from the original inference rules (Fig. 5.5) apply to the inference rules for any monotonic analysis. Many such analyses exist (the dataflow-problem abstraction is the canonical means of describing a compiler analysis), and many of these analyses compute program properties that can also be checked and propagated at runtime. The main challenge is to determine how to perform checks, track check results, and recover from check failures.

This line of work could be seen as a type of *specialization*, in which a compiler transforms code in a way that applies to a subset of all possible inputs and dynamically chooses this code. The advantage of our approach is that it outlines a mechanical way of deriving such an analysis, easing its wider adoption.

6.1.4 General Cost/Benefit Compiler-Experiment Framework

In the course of developing both Daedalus and Icarus, we made several heuristic decisions that have a potentially large hand in performance: in particular, which loops to parallelize, and which dynamic checks to insert. The heuristics that we used were simple (for Daedalus, profiling-based; for Icarus, parallelize as many loops as possible, and push checks as early as possible). However, many more complex types of reasoning are possible. The conventional approach of developing cost and benefit functions (such as is commonly used for function-inlining decisions) might work well, although the benefit of loop parallelization can be somewhat hard to predict in the presence of inserted locking and without profiling information about loop trip counts and iteration-length distributions.

We envision a more general framework that can make *hypotheses* and *backtrack* at multiple levels, both in analysis and potentially during runtime (using a JIT-like framework that permits transforms during program execution). At static-analysis time, the use of backtracking could enable better placement of dynamic checks, for example. During runtime, the system could hypothesize a certain benefit, perform a transform, and revert this decision and take a different path if dynamic checks fail too often or performance is not as expected. In general, the combination of dynamic experimentation and profiling with the use of dynamic checks and hybrid static-dynamic analysis appears to be a very fruitful ground for the development of new ideas and techniques.

6.1.5 Full Compiler Backend for Loop Parallelization

We evaluated loop parallelization in this work using simulation in order to enable the study of varying parameters, and to avoid the significant engineering overhead of building and optimizing a real compiler-transform backend. We believe this was a prudent decision given the time and resources available and the relative payoffs of different approaches in terms of research conclusions and new ideas. However, the DAEDALUS/ICARUS system should be extended with a robust, well-optimized backend that actually parallelizes programs in order to complete the toolchain. In fact, we prototyped such a backend in earlier loop-parallelization work (based on data-structure-aware dynamic analysis), but it needs significant optimization in order to produce satisfactory results, due to many of the same overheads described in Chapter 5.

6.1.6 Analysis of Systems Languages Such As C/C++

Although our system is not fundamentally tied to Java as an input language (its principles apply to any imperative language), Java does allow for some simplifying assumptions w.r.t. heap behavior in particular that are not present in all languages. For example, in common systems programming languages such as C and C++, pointers can alias the internals of a data structure (individual fields or sub-structs), and arbitrary aliasing and type-punning can occur via the use of unions or arbitrary pointer casting. Points-to analyses exist for C/C++, so solving these issues at a basic level is not new ground. However, some thought would be required to add the data-structure primitives and to model standard library types such as the C++ STL with them. Finally, on a practical level, lower-level languages such as C do not have standard-library facilities for common data structures, leading to the existence of many third-party libraries and many one-off implementations of lists, hash-tables, and other types. Additional work, or perhaps a new type of analysis, might be necessary to enable semantic models to capture the behavior of these data types in such a language in a practical way.

6.1.7 Applications of Distinctness to Type Systems

We believe that the distinctness concept could be applied to enhance a type system's description of the program's heap in ways that would be helpful both in ensuring program correctness and in enabling parallelization. This could be similar in concept to the region/partition-based system in Legion [107], in that it describes a partitioning of the heap that is based on the program's data structure. However, the analysis abilities of Daedalus extend beyond those of Legion by virtue of its first-class high-level data structures. The analysis performed by Daedalus then becomes akin to type inference, and the programmer can help the system along by explicitly annotating distinctness on pointer types as necessary. Furthermore, distinctness could become part of an API contract: for example, a library-level parallel-loop construct that takes user code as a first-class function could specify distinctness requirements on all pointers passed into that function.

6.2 Conclusion

In this thesis, we first introduced IR (intermediate representation) primitives in the compiler for the common map, list, and set data structures, and semantic models that map portions of the program or standard library to use these primitives directly. We showed that this change enables the compiler to analyze program heap behavior and aliasing with significant additional precision. We then described Daedalus (Data-structure-aware Distinctness Analysis), a new type of alias analysis that discovers *distinctness*, or non-aliasing across loop iterations, in local program variables and related non-aliasing invariants on the program heap. We provided a set of rules to determine when a loop is parallelizable based on distinctness, and we demonstrated that this loop-parallelization system achieves significantly better speedup on a set of Java benchmarks than prior array-indexing-based systems. Finally, we addressed analysis limitations of Daedalus by introducing Icarus (Integrated Compiler and Runtime with User-level Semantics), a system that incorporates *dynamic checks* to augment the static conclusions of Daedalus in a principled way. We first developed a set of general principles to convert the static-analysis rules into rules that reason about hybrid static/dynamic facts and the best locations to perform dynamic checks. We then described a runtime system that is able to execute parallelized loops with dynamic checks, *without* any speculation or rollback upon check failure, in a sound manner. We showed that Daedalus achieves significant additional loop parallelization and hence speedup over Icarus and prior work.

We believe that this system may serve as a foundation for additional work on high-level program analysis and transformation by providing a means to reason about program behavior more precisely and by exposing the higher-level operations that are usually not visible to an analysis buried in the details of their implementations. We hope that work continues on this promising approach.

Appendix A

Definitions and Proofs for Daedalus

In this appendix, we provide a more detailed specification of the inference rules in the Daedalus analysis, as well as a soundness proof. We will show first that the analysis derives correct distinctness judgments, and then that if a loop is parallelized according to distinctness-based conditions that we give, then its iterations will execute correctly.

A.1 Definitions

We analyze a program P consisting of methods in classes. Each method M_i has a control-flow graph (CFG) of statements S_i in basic blocks B_i. We assume the program is in SSA (static single assignment) form. The two requirements on the program control flow are: (i) no irreducible loops exist (this is true by construction for JVM bytecode emitted by javac, whose CFG corresponds to the original Java code's structured control flow); and (ii) main() does not participate in a corecursive cycle in the call graph.

A given program may produce any trace T_i of a (possibly infinite) family of execution traces T. Each trace is a sequence of *dynamic instances* of static statements; each dynamic instance is annotated with the value(s) produced by that statement. We notate dynamic instance j of statement S_i in execution trace T as $T[i, j]$.

The program operates on a heap consisting of scalar objects with fields, and map objects indexed by object identity (pointer value). Every field and map slot stores either a primitive value or an object reference (pointer).

We assume a may-point-to analysis has run, producing a points-to set consisting of *heap abstractions* for each variable definition in the (static) program. Each heap object during any execution of the program corresponds to exactly one heap abstraction.

A.2 Loops and Loop Contexts

We first find the natural loops in each method. For a given basic block B_i, the block is contained in the body of zero or more loops $L(B_i) = L_1, L_2, \ldots$. Loops are organized into a loop-nest tree. We analyze programs with no irreducible loops (compiled from Java, which has structured control-flow), hence every backedge in a method's CFG must be a backedge of some loop.

A loop that is identified statically has *dynamic instances* in any given execution trace, and each instance has *iterations.* Any given dynamic statement instance $T[i, j]$ in execution trace T that is contained in a loop L_k has a loop instance number $\text{Inst}[i, j, k]$ and loop iteration number within that instance of $\text{Iter}[i, j, k]$. Two distinct dynamic instances must differ in either instance number or iteration number: if the path between them crosses the loop backedge but does not exit the loop body, then the iteration number must differ; if the path exits the loop body, then the instance number must differ.

We define the *loop context* of a basic block B_i, $\mathbb{L}(B_i)$, as a set that is initially its containing loop set $L(B_i)$. We overload this notation to say that $\mathbb{L}(S_i)$ is the loop context of S_i. Thus, if S_i is in basic block B_i, $\mathbb{L}(S_i)$ is the same as $\mathbb{L}(B_i)$.

Lemma 1. *Any two dynamic instances $T[i, j]$ and $T[i, j']$ of statement S_i ($j \neq j'$) in execution trace T that occur in the same method invocation of containing method M occur in different iterations of the same instance of some loop L_k in the loop context of S_i: $Inst[i, j, k] = Inst[i, j', k]$ and $Iter[i, j, k] \neq Iter[i, j', k]$.*

Proof. For two dynamic instances of a single statement to occur in one method invocation, the path between them must flow across a backedge of at least one loop k within the method. Either $\text{Inst}[i, j, k] \neq \text{Inst}[i, j', k]$ or $\text{Iter}[i, j, k] \neq \text{Iter}[i, j', k]$

Case I. If $\text{Inst}[i, j, k] = \text{Inst}[i, j', k]$, then it must be the case that $\text{Iter}[i, j, k] \neq \text{Iter}[i, j', k]$, and we are done.

Case II. If $\text{Inst}[i, j, k] \neq \text{Inst}[i, j', k]$, then the path exits and re-enters the body of L_k. This can only occur by crossing the backedge of some other loop $L_{k'}$ (any edge from a block following an exit of L_k to a block preceding the header of L_k forms another natural loop) which is strictly larger than L_k (contains the body of L_k plus at least one other block). Then consider this argument where k' becomes k. Because every such recursive step of this argument considers a strictly larger loop L_k, we cannot consider a given loop twice; because there are finitely many loops in a method, we must eventually reach Case I above for some k, and we are done. □

We then augment the loop context set of a block with loop(s) representing any possible repetition of the method containing that block: the *method loop context.* There are two cases: first, if a method (qualified with static analysis context) is called from only one callsite, its method loop context is simply the loop context of that callsite. Second, if a method is called from more than one callsite, its method loop context consists of the intersection of all callsites' loop contexts as well as a new *method repeat loop*. The latter loop is a conceptual entity that exists only during analysis; it is treated as if it has only one instance during execution, and its iteration number increments with each call to the method. (This is never actually implemented at runtime; it is only assumed behavior for analysis.)

Lemma 2. *Any two dynamic instances $T[i,j]$ and $T[i,j']$ of statement S_i ($j \neq j'$) in execution trace T must have differing loop instance numbers $Iter[i,j,k] \neq Iter[i,j',k]$ for some loop L_k in the loop context of S_i where $Inst[i,j,k] = Inst[i,j',k]$. Informally, we say that* the loop context captures all repetition.

Proof. We argue this property inductively over strongly-connected components of the callgraph, in a particular order: from caller SCCs to callee SCCs, and in a given SCC, all methods with multiple callers first (which must include all methods called by callers outside the SCC), then all methods with only one caller, only after their callees are visited (every such method must eventually be visited because a chain of single-caller methods in an SCC must eventually be called by a method with more than one caller, or else they would form an unreachable cycle). This ensures that any single-caller method can rely on the lemma (inductive hypothesis) having been proven for its caller, which we use in Case II-a below. No other case makes use of this inductive hypothesis.

Case I. The two instances are in the same method invocation of method M of statement S_i (i.e., the path P from $T[i,j]$ to $T[i,j']$ does not pass through a return instruction in M). By Lemma 1, there exists a k such that $\text{Inst}[i,j,k] = \text{Inst}[i,j',k]$ and $\text{Iter}[i,j,k] \neq \text{Iter}[i,j',k]$.

Case II. The two instances are in different invocations of method M. Then the path P passes through a return instruction in M, and subsequently through some callsite of M.

Case II-a. Method M has exactly one callsite. $T[i,j]$ and $T[i,j']$ must have occurred in invocations from two dynamic instances of the callsite $T[p,q]$ and $T[p,q']$. By the inductive hypothesis, there must be some loop L_k satisfying the required condition for these two callsite instances. This loop will be in the loop context of M (by construction) and thus S_i, so we are done.

Case II-b. Method M has more than one callsite. Then there is a unique method repeat loop L_k in the method loop context of M. The two instances $T[i,j]$ and $T[i,j']$ then have $\text{Inst}[i,j,k] = \text{Inst}[i,j',k]$ and $\text{Iter}[i,j,k] \neq \text{Iter}[i,j',k]$. by construction. □

We have thus shown that the loop context of any statement captures all repetition: intuitively, any repeat execution of this statement must be due to the backedge of some loop in context.

$$\frac{[v_i \;:=\; \text{new } T]_i}{\text{Tag}[v_i] = \{\text{Alloc}_i\} \quad \text{Orig}[\text{Alloc}_i] = S_i \\ \text{TagDepth}[\text{Alloc}_i] = 1} \; [\textsc{TagAlloc}]$$

$$\frac{[v_i \;:=\; v_j]}{\text{Tag}[v_i] = \text{Tag}[v_j]} \; [\textsc{TagAssignSingle}]$$

$$\frac{[v_i \;:=\; \phi(v_{j_1}, v_{j_2}, \ldots, v_{j_n})]_i \qquad \forall k, l.\text{pts}(v_{j_k}) \cup \text{pts}(v_{j_l}) = \emptyset}{\text{Tag}[v_i] = \{\text{Assign}_i\} \quad \text{Orig}[\text{Assign}_i] = S_i \\ \text{TagDepth}[\text{Assign}_i] = 1} \; [\textsc{TagAssignMultiNonOverlap}]$$

$$\frac{[v_i \;:=\; \phi(v_{j_1}, v_{j_2}, \ldots, v_{j_n})]_i \qquad \exists k, l.\text{pts}(v_{j_k}) \cup \text{pts}(v_{j_l}) \neq \emptyset}{\text{Tag}[v_i] = \cup_{m=1}^{n} \text{Tag}[v_{j_m}]} \; [\textsc{TagAssignMultiOverlap}]$$

$$\frac{[v_i \;:=\; v_j.f]_i \qquad \text{FieldConst}(f) \qquad t \in \text{Tag}[v_j] \qquad \text{TagDepth}[t] < \text{DepthLimit}}{\text{Tag}[v_i] = \{t.f\} \quad \text{Orig}[t.f] = \text{Orig}[t] \\ \text{TagDepth}[t.f] = \text{TagDepth}[t] + 1} \; [\textsc{TagLoadConstField}]$$

$$\frac{[v_i \;:=\; v_j.f]_i \qquad \textit{not otherwise handled by rules above}}{\text{Tag}[v_i] = \{\text{Other}_i\} \quad \text{Orig}[\text{Other}_i] = S_i \\ \text{TagDepth}[\text{Other}_i] = 1} \; [\textsc{TagOther}]$$

(We pre-compute the predicate FieldConst(f) to hold for any field f that is never mutated after object initialization. This is useful to know because we can rely on the fact that if we have two such objects v_1, v_2 such that $v_1 = v_2$, then we know $v_1.f = v_2.f$.)

Figure A.1: Inference rules for tag-based must-alias analysis.

A.3 Tag-based Pseudo-flow-sensitive Must-Alias Analysis

Next, we define a tag-based must-alias analysis, and show that it provides a useful aliasing property.

We define Tag[v_i] to be a set of symbols (tags) associated with the variable v_i defined at statement S_i. Intuitively, tags identify data flow of pointers through the program, and become useful in reasoning about aliasing between different pointers with the help of distinctness facts.

Every tag has an *originator statement*, S_{tag}, which we define explicitly with tag-propagation rules below alongside the tag sets for each variable.

Let us define the meaning of tags on variables. First, we consider a slightly simplified *single-tag* version of the analysis, where each tag set Tag[v_i] has at most one element. We can imagine each originator statement to be a write to a dynamic storage slot corresponding to that tag, and every other tagged variable as a read of that particular tag's slot and then an equality assertion between the read value and the value of the tagged variable. For example, if S_1 defines v_1, v_1 is tagged with tag t (for which S_1 is the originator statement), and S_2 defines v_2 also tagged with t, then if in a particular execution S_1 executes and assigns values x, y, z to v_1 in turn, then S_2 executes, v_2 must take on value z (the most recent value produced by the originator statement).

Now consider tag sets with multiple elements. In this case, a statement S_i assigning variable v_i with tags $t_1, t_2, \ldots, t_n$ must assign a value V to v_i that is equal to the last assigned value at *one of* the originator statements for these tags.

Formally, we say that if a statement S_i defines v_i with tag set $\text{Tag}[v_i] = \{t_1, \ldots, t_n\}$ then for any given dynamic instance $T[i, j]$ of S_i, there must exist some dynamic instance $T[t, u]$ of an originator statement S_t assigning to v_t with tag t_k $(1 \leq k \leq n)$ such that $v_{t,u} = v_{i,j}$, and the dynamic instance $T[t, u]$ is the latest instance of S_t prior to $T[i, j]$.

A.3.1 Tag Propagation

We next define a set of inference rules that fill the tag-sets on every variable, with each statement type treated in its own way. Fig. A.1 shows these rules.

A remark on notation: we will denote a program statement S_i as (e.g.) $[v_i := \text{new } T]_i$, and we will denote inference rules here with the standard horizontal line.

We will denote the originator of a tag t_i as $\text{Orig}[t_i]$. Furthermore, we denote the points-to set from a may-point-to analysis as $\text{pts}(v_i)$. When two points-to sets have a non-null intersection, this simply means that some dynamic instance of v_i may be equal to some dynamic instance of v_j. Conversely, if the points-to sets do not intersect, then no dynamic instance of v_i may ever be equal to a dynamic instance of v_j in any execution.

A.3.2 Tags: Properties and Proofs

Aliasing Implied by Tags:

We next prove that the propagation rules above produce tag sets that are consistent with the definition above: informally, that tags correspond to program dataflow and certain must-alias relations.

We state the main tag-aliasing lemma as follows:

Lemma 3. *Given a variable v_i defined at S_i, with value $v_{i,p} = V$ at dynamic instance $T[i, p]$, if Tag$[v_i] = \{t\}$ (exactly one element) and if S_i is not the originator of t, then the value V is equal to the value assigned (not necessarily to v_i) by the latest dynamic instance $T[j, q]$ of the tag-originator statement Orig$[t]$ prior to $T[i, p]$ in T.*

Proof. We can proceed inductively along an execution trace T, showing this property holds for each dynamic statement instance in turn. Setting aside the [TagLoadConstField] rule momentarily, we consider all statements that propagate (rather than originate) tags: these are assignments with one or more inputs. We can step backward in the execution trace through tag-propagating statements to find the set of dynamic

$$\frac{[v_i\ := \phi(\ldots, v_{j_k}, \ldots)]_i}{\text{Assign}(S_i, v_i, v_{j_k})}\ [\text{IntraproceduralAssign}]$$

$$\frac{\begin{array}{ll}[v_i\ := \text{call } v_o.M(v_{a_1}, v_{a_2}, \ldots)]_i & [v_j = \text{arg}_k]_j \\ \text{CallGraphEdge}(S_i, M_{\text{callee}}) & S_j \in M_{\text{callee}}\end{array}}{\text{Assign}(S_i, v_j, v_{a_k})}\ [\text{AssignParamValue}]$$

$$\frac{\begin{array}{ll}[v_i\ := \text{call } v_o.M(v_{a_1}, v_{a_2}, \ldots)]_i & [v_j = \text{this}] \\ \text{CallGraphEdge}(S_i, M_{\text{callee}}) & S_j \in M_{\text{callee}}\end{array}}{\text{Assign}(S_i, v_j, v_o)}\ [\text{AssignThisValue}]$$

$$\frac{\begin{array}{ll}[v_i\ := \text{call } v_o.M(v_{a_1}, v_{a_2}, \ldots)]_i & [\text{return } v_r]_j \\ \text{CallGraphEdge}(S_i, M_{\text{callee}}) & S_j \in M_{\text{callee}}\end{array}}{\text{Assign}(S_i, v_i, v_r)}\ [\text{AssignReturnValue}]$$

$$\frac{\text{Assign}(S_i, v_i, v_j) \qquad \text{NotDistinct}(v_j, L) \qquad L \in \mathbb{L}(S_i)}{\text{NotDistinct}(v_i, L)}\ [\text{AssignNotDistinct}]$$

$$\frac{L = \text{InnermostLoop}[S_i] \qquad L' \subset L \qquad \text{Assign}(S_i, v_i, v_j)}{\text{NotDistinct}(v_i, L')}\ [\text{AssignNotDistinctInSubLoop}]$$

$$\frac{|\text{Tag}[v_i]| > 1 \qquad L \in \text{MethodLoops}[S_i]}{\text{NotDistinct}(v_i, L)}\ [\text{NotDistinctIfTagConflict}]$$

Figure A.2: Distinctness inference rules for assignment statements.

instances of tag-originator statements TOS $= \{S_{o_1}, \ldots\}$. As the lemma applies only to variables with one tag ($\text{Tag}[v_i] = \{t\}$), and as all tags are propagated to v_i, we must have only one tag-originator statement TOS $= \{S_o\}$. We must simply show that the value $v_{o,q}$ assigned to v_o by the last dynamic instance $T[o, q]$ of S_o is the one that reaches $v_{i,p}$. To show this, we can simply see that S_o must dominate S_i for the SSA to be well-formed, and because v_i can trace back only to v_o via assignments, any path from S_o to S_i must carry the value from this instance of S_o to S_i. Hence, in any path from S_o to S_i, any additional instance of S_o between the endpoints would have become the instance of S_o that fed to v_i at S_i.

Now consider loads from constant-once-constructed fields. Soundness of [LoadConstField] follows in a straightforward way: because by definition of the FieldConst predicate, $x.f$ always returns the same value y for a given x, we can conclude that if v_j at S_j has tag t implying equality to the last dynamic instance of v_o at originator at S_o, then it is valid to describe the field $v_j.f$ at S_j with tag $t.f$, equal to $v_o.f$ at originator S_o. $\square$

$$\frac{\text{MethodRepeatLoop}(L) \qquad L \in \mathbb{L}(S_i) \qquad [v_i \ := \text{arg}_k]_i \vee [v_i \ := \text{this}]_i}{\text{NotConstant}(v_i, L) \quad \text{NotDistinct}(v_i, L)} \text{ [ParamsRepeatLoop]}$$

$$\frac{L \in \text{MethodLoops}(S_i) \qquad L \notin \mathbb{L}(S_i) \qquad [v_i \ := \ldots]_i}{\text{NotDistinct}(v_i, L)} \text{ [NotDistinctInLoopsNotInContext]}$$

$$\frac{L \in \text{MethodLoops}(S_i) \qquad [v_i \ := \ldots]_i \qquad \text{NotHandled}(S_i)}{\text{NotDistinct}(v_i, L)} \text{ [DefaultNotDistinct]}$$

Figure A.3: Auxiliary rules: handling of method-repeat loops, and production of not-distinct judgments for all cases not covered by other rules.

$$\frac{\begin{array}{lll} \neg[v_i & := & v_j]_i \\ \neg[v_i & := & \phi(\ldots)]_i \\ \neg[v_i & := & \text{Int\_Constant}]_i \\ \neg[v_i & := & v_j.f]_i \end{array} \qquad \begin{array}{l} L = \text{InnerLoop}[S_i] \\ L \subseteq L' \end{array}}{\text{NotConstant}(v_i, L')} \text{ [DefaultNotConstant]}$$

Figure A.4: Rules for variable constantness derivations.

A.4 Distinctness Analysis

Finally, we introduce *distinctness analysis*. Distinctness analysis produces a body of knowledge about cross-iteration aliasing of variables with respect to loops in a program. We also define the related concept *constantness*, useful in deriving invariants on heap objects.

A.4.1 Variable Distinctness

We define a variable v_i to be *distinct* with respect to loop L_k for any two dynamic instances $T[i,p]$ and $T[i,q]$ of S_i where $\text{Inst}[i,p,k] = \text{Inst}[i,q,k]$ and $\text{Iter}[i,p,k] \neq \text{Iter}[i,q,k]$, the variable values defined at these instances are not equal: $v_{i,p} \neq v_{i,q}$. In other words, v_i *never aliases across iterations*. We write $\text{Distinct}(v_i, L)$ to denote the judgment that v_i is distinct with respect to L.

We make one further refinement to simplify the system. As we will see below, distinctness analysis involves many *intersections* of distinctness properties at meet-points, and so in a monotonic analysis (such as one written in Datalog), it is more natural to derive *non-distinctness* facts. We will write inference rules here to derive non-distinctness, with the understanding that we can recover distinctness as follows: v_i is distinct w.r.t. L_k if L_k is in the loop context of any statement of the method defining v_i, and there is no not-distinct fact for v_i w.r.t. L_k. (That is, the universe is the set of all loops in the method, and not-distinct judgments descend down the lattice from $\top$, the full universe of distinctness facts, toward $\bot$, the empty set.)

Fig. A.2 provides the inference rules to produce not-distinct judgments on variable assignment statements (and implicit assignments of parameter and return values across callgraph edges). We now prove the following lemma:

Lemma 4. *Distinctness inference rules triggered by assignment statements produce correct distinctness facts on output variables.*

Proof. Follows directly from program semantics. If an assignment source (variable) has a distinctness fact with respect to a loop L, then it must be defined inside the body of L; because uses of a variable in SSA must be dominated by the variable definition, this use cannot cross the loop backedge, so it must originate in the current iteration. There are thus two ways for a variable to alias across iterations of a loop when produced by this assignment: a single assignment source yields the same value on two different iterations (Case I) or the assignment has multiple possible sources (a ϕ-node) and the same value is yielded by two of the sources in different iterations (Case II). Hence not-distinct judgments are propagated across all assignments by [AssignNotDistinct], or created by [NotDistinctIfTagConflict] when two different values (identified by tags) meet at a ϕ-node, respectively. □

We can also now prove a useful lemma that allows us to extend the distinctness invariant from the point of definition (assignment) to the point of use:

Lemma 5. *Given a variable v_i defined at S_i and used at S_j, for any loop $L \in (\mathbb{L}(S_i) \cap \mathbb{L}(S_j))$, if Distinct$(v_i, L)$, then the use of v_i at S_j reads a distinct value at every iteration of L.*

Proof. We can easily see that this is true by creating a virtual assignment of a new variable $[v_{i'} := v_i]_j$ at each use site S_j of v_i. By the assignment distinctness-propagation rules and corresponding argument above, $v_{i'}$ has exactly the same distinctness facts as v_i for all loops in context both at the definition site S_i and use site S_j. □

A.4.2 Miscellaneous and Fallback Rules

The rules in Fig. A.3 simply ensure that all values produced by statements not otherwise handled by these inference rules are marked as not-distinct in all relevant loops. In addition, [ParamsRepeatLoop] ensures that method parameters are considered not-distinct across multiple invocations of the method, if the creation of a method-repeat loop is necessary to preserve Lemma 2.

A.4.3 Object Allocation

The rule in Fig. A.5 covers object allocation statements, and simply states that an allocation is not-distinct in any subloops of the inner loop of the allocation site; hence, by omission, it *is* distinct in all loops in context. This follows from the semantics of object allocation: each allocated object does not alias with any other still-reachable allocation.

$$\frac{[v_i \ := \text{new } T]_i \qquad L = \text{InnerLoop}[S_i] \qquad L' \subset L}{\text{NotDistinct}(v_i, L')} \ [\textsc{AllocNotDistinctInSubLoop}]$$

Figure A.5: Rule for object allocations: an allocation is distinct w.r.t. all containing loops of the allocation site.

$$\frac{[x_1.f := y_1]_i \qquad [x_2.f := y_2]_j \qquad i \neq j \qquad X \in (\text{pts}(x_1) \cap \text{pts}(x_2))}{\text{FieldNotDistinct}(X.f)} \ [\textsc{StoreOverlap}]$$

$$\frac{\begin{array}{lll} [x_1.f := y_1]_i & \text{NotConstant}(x_1, L) & \\ L \in \mathbb{L}(S_i) & \text{NotDistinct}(y_1, L) & X \in \text{pts}(x_1) \end{array}}{\text{FieldNotDistinct}(X.f)} \ [\textsc{FieldNotDistinct}]$$

Figure A.6: Rules for stores to object fields.

A.4.4 Variable Constantness

The inference rules in Fig. A.4 compute the *constantness* of variables in the program with respect to its loops. They follow directly from the semantics of the program statements. Variables are not constant with respect to loops not in their context. Assignments propagate not-constant facts. A load can propagate constantness (and thus not-constantness) if, as above, we have a FieldConst(f) judgment on the loaded field f. Finally, as for distinctness above, a tag-conflict forces a not-constant judgment with respect to every loop in context because merging two value sources at a ϕ-node eliminates any guarantees we could make about either one of the sources alone.

A.4.5 Field and Map Distinctness

Next, we define *field distinctness* and its extension to map objects, *map distinctness*, so that stores and loads can propagate distinctness invariants into and out of the heap, respectively.

We say that a field f on a points-to abstraction X – denoted $X.f$ – is *distinct* if for any two $x_1, x_2 \in X$, $x_1 \neq x_2 \Rightarrow x_1.f \neq x_2.f$.

We say that an (M, K)-tuple of heap abstractions, where M represents map heap abstractions and K represents key abstractions that index those maps, is *globally map-distinct* if for any $m_1, m_2 \in M$, and any two $k_1, k_2 \in K$, $m_1 \neq m_2 \vee k_1 \neq k_2$ implies $m_1[k_1] \neq m_2[k_2]$. We say that the (M, K)-tuple is *within-map distinct* if instead $m_1 = m_2 \wedge k_1 \neq k_2$ implies $m_1[k_1] \neq m_2[k_2]$.

Stores: We first infer which fields are distinct. Fig. A.6 shows the rules that apply to all field stores to derive these judgments.

Lemma 6. *If a field f on X is distinct as inferred by the [*StoreOverlap*] and [*FieldNotDistinct*] rules, then for any $a, b \in X$, $a \neq b$ implies $a.f \neq b.f$.*

$$\frac{[x_1 := y_1.f]_i \qquad \text{NotDistinct}(y_1, L)}{\text{NotDistinct}(x_1, L)} \text{[LoadBaseNotDistinct]}$$

$$\frac{[x_1 := y_1.f]_i \qquad Y \in \text{pts}(y_1) \qquad \text{FieldNotDistinct}(Y.f) \qquad L \in \mathbb{L}(S_i)}{\text{NotDistinct}(x_1, L)} \text{[LoadFieldNotDistinct]}$$

$$\frac{\begin{array}{c}[x_1 := y_1.f]_i \\ Y_1 \neq Y_2\end{array} \qquad \begin{array}{c}Y_1 \in \text{pts}(y_1) \\ Y_2 \in \text{pts}(y_1) \\ X \in \text{pts}(Y_1.f) \\ X \in \text{pts}(Y_2.f)\end{array} \qquad L \in \mathbb{L}(S_i)}{\text{NotDistinct}(x_1, L)} \text{[LoadConflict]}$$

Figure A.7: Rules for loads from object fields.

Proof. We first note that only one store instruction stores values to field f of any $x \in X$, because if any other store also wrote to the same field and abstraction, [StoreOverlap] would produce a FieldNotDistinct$(X.f)$ judgment. Let us thus consider this one store $[x_1.f := y_1]_i$.

Let us assume that the store breaks the field-distinctness invariant by storing a single value to the field on two different instances: $a \neq b$ and $a.f = b.f$, yet the [FieldNotDistinct] rule does not conclude FieldNotDistinct$(X.f)$. Then there are two different dynamic instances $T[i,p]$ and $T[i,q]$ of S_i, such that y_1 at $T[i,p]$ is equal to y_1 at $T[i,q]$ but x_1 at $T[i,p]$ is not equal to x_1 at $T[i,q]$. But for the rule not to apply, we must have either no NotConstant(x_1, L) judgment (hence the base pointer x_1 is constant) or no NotDistinct(y_1, L) judgment (hence the stored value y_1 is distinct) for every $L \in \mathbb{L}(S_i)$. By Lemma 2, any two dynamic instances of the store are in separate iterations of the same instance of some loop L in the loop context. Hence for these two instances that cause the field aliasing, either the base pointers x_1 must be equal or the stored values y_1 must not be equal. This contradicts the above; thus the rule must apply (and produce a FieldNotDistinct$(X.f)$ judgment) whenever the field is not distinct, and so the absence of such a judgment implies field distinctness. □

Loads: Next, we introduce the inference rules that use field distinctness to produce variable distinctness judgments at load statements. These rules are given in Fig. A.7.

Lemma 7. *The three rules in Fig. A.7 correctly derive distinctness of the result of an object field load with respect to all loops in context.*

Proof. The rules must produce a NotDistinct(x_1, L) judgment whenever two instances of the load in two separate iterations of one instance of loop L produce the same value. There are two ways in which this could happen: (i) the same base pointer value in y_1 is provided to each instance, thus yielding the same loaded value if no intervening store has modified the field, or (ii) two different objects hold the same value in their

$$\frac{\begin{array}{l}[\text{mapput } x_1, y_1, z_1]_i \\ [\text{mapput } x_2, y_2, z_2]_j\end{array} \quad i \neq j \quad \begin{array}{l} X \in (\text{pts}(x_1) \cap \text{pts}(x_2)) \\ Y \in (\text{pts}(y_1) \cap \text{pts}(y_2))\end{array}}{\text{MapNotDistinct}(X[Y]) \quad \text{MapNotDistinctWithinMap}(X[Y])} \; [\textsc{MapStoreOverlap}]$$

$$\frac{\begin{array}{ll}[\text{mapput } x_1, y_1, z_1]_i & \\ X \in \text{pts}(x_1) & \text{NotDistinct}(z_1, L) \\ Y \in \text{pts}(y_1) & \text{NotConstant}(x_1, L) \vee \text{NotConstant}(y_1, L)\end{array}}{\text{MapNotDistinct}(X[Y])} \; [\textsc{MapStoreNotDistinct}]$$

$$\frac{\begin{array}{ll}[\text{mapput } x_1, y_1, z_1]_i & \\ X \in \text{pts}(x_1) & \text{NotDistinct}(z_1, L) \\ Y \in \text{pts}(y_1) & \text{NotConstant}(y_1, L)\end{array}}{\text{MapNotDistinctWithinMap}(X[Y])} \; [\textsc{MapStoreNotDistinctWithinMap}]$$

Figure A.8: Rules for map stores.

$$\frac{\begin{array}{ll} & \text{MapNotDistinct}(X[Y]) \vee \\ [z_1 := \text{mapget } x_1, y_1]_i & \quad (\text{NotDistinct}(x_1, L) \wedge \\ X \in \text{pts}(x_1) & \quad\quad \text{NotDistinct}(y_1, L)) \\ Y \in \text{pts}(y_1) & \text{MapNotDistinctWithinMap}(X[Y]) \vee \\ & \quad \text{NotConstant}(x_1, L) \vee \\ & \quad \text{NotDistinct}(y_1, L)\end{array}}{\text{NotDistinct}(z_1, L)} \; [\textsc{MapLoadNotDistinct}]$$

$$\frac{\begin{array}{ll}[z_1 := \text{mapget } x_1, y_1]_i & \\ X_1 \in \text{pts}(x_1) & X_1 \neq X_2 \vee Y_1 \neq Y_2 \\ Y_1 \in \text{pts}(y_1) & Z \in \text{pts}(X_1[Y_1]) \\ X_2 \in \text{pts}(x_1) & Z \in \text{pts}(X_2[Y_2]) \\ Y_2 \in \text{pts}(y_1) & L \in \mathbb{L}(S_i)\end{array}}{\text{NotDistinct}(z_1, L)} \; [\textsc{MapLoadConflict}]$$

Figure A.9: Rules for map loads.

instances of field f. In the first case, [LoadBaseNotDistinct] will produce the appropriate not-distinct result judgment. In the second case, either the field was not distinct (and the [LoadFieldNotDistinct] rule will produce a not-distinct judgment), or the pointer y_1 visited objects represented by two different heap abstractions, such that even if the field f were distinct on each, it could alias across the different abstractions (the [LoadConflict] rule will ensure correctness in this case). □

Map Stores: We give three rules to derive the two types of map-related distinctness in Fig. A.8. These rules are analogous too (in fact, a generalized form of) the rules for ordinary field stores.

Lemma 8. *The rules in Fig. A.8 correctly derive map distinctness.*

Proof. Analogous to the store rules above, with the following extensions. A map value slot on the heap is identified by two values, the map and the key, rather than one as for object fields (the base pointer).

Thus, when we show *global map-distinctness* (no aliasing across any two map slots for any two maps represented by the map heap abstraction), the constant-base-pointer requirement (if stored value is not distinct) becomes "constant map and constant key." In other words, we violate global map distinctness if we store the same value to the same key in two different maps, or the same map at two different keys, or two different maps at two different keys. The negation of this is not-constant-map or not-constant-key, as seen in the rule [MapStoreNotDistinct].

Likewise, when we show *within-map distinctness* (no aliasing across any two slots in a single map represented by the map heap abstraction), the constant-base-pointer requirement (if stored value is not distinct) becomes simply "constant key." In other words, we only violate within-map distinctness if we store the same value to different keys. The negation of this is simply not-constant-key, as seen in the rule [MapStoreNotDistinctWithinMap]. □

Map Loads: Finally, we provide a rule to derive distinctness judgments on the results of map loads. Fig. A.9 provides this rule.

Lemma 9. *The rules in Fig. A.9 correctly derive distinctness of the result of a map load with respect to all loops in context.*

Proof. The result can be shown *distinct* in two different ways, with the aid of either global map distinctness or within-map distinctness. A load from a map on which we have shown global map distinctness produces a distinct value with respect to a loop if either the map or the key (or both) is distinct each iteration: global map distinctness implies that changing either of the two components of the map-slot address will result in a different value. Additionally, if the map values have within-map distinctness, then a distinct key *and a constant map* together will yield differing values.

The rules above are simply the negation of these conditions. The result will be not-distinct if neither of the above conditions can show it to be distinct. Hence, either there is no global map distinctness, or both the map and key are not-distinct; and, either there is no within-map distinctness, or the map is not constant, or the key is not distinct. If the above conditions do not yield a not-distinct result for any loop in context, then the result must be distinct.

Finally, in the above we assumed that non-distinctness judgments on map contents applied across all visited objects, while in actuality they apply only to one (M, K)-tuple of a map and key heap abstraction. We bridge this gap with the [MapLoadConflict] rule (analogous to the [LoadConflict] rule in the scalar field load case) that produces a not-distinct judgment whenever two or more (M, K)-tuples can point to the same heap abstraction. □

$$\frac{\begin{array}{ll} \text{StatementInLoop}(S_i, L) & [x_i.f := y_i]_i \vee [y_i := x_i.f] \\ \text{HasStoresInLoop(X.f, L)} & X \in \text{pts}(x_i) \\ & \text{NotDistinct}(x_i, L) \end{array}}{\text{NotParallelizable}(L)} \quad [\textsc{LoopFieldAccessNotDistinct}]$$

$$\frac{\begin{array}{ll} \text{StatementInLoop}(S_i, L) & [x_i.f := y_i]_i \vee [y_i := x_i.f] \\ \text{StatementInLoop}(S_j, L) & [x_j.f := y_j]_j \vee [y_j := x_j.f] \\ \text{HasStoresInLoop(X.f, L)} & X \in (\text{pts}(x_i) \cap \text{pts}(x_j)) \\ & |\text{Tag}[x_i] \cup \text{Tag}[x_j]| > 1 \end{array}}{\text{NotParallelizable}(L)} \quad [\textsc{LoopFieldAccessTagConflict}]$$

$$\frac{\begin{array}{ll} & [x_i[y_i] := z_i]_i \vee [z_i := x_i[y_i]] \\ & X \in \text{pts}(x_i) \\ \text{StatementInLoop}(S_i, L) & Y \in \text{pts}(y_i) \\ \text{HasStoresInLoop(X[Y], L)} & \text{NotDistinct}(x_i, L) \\ & \text{NotDistinct}(y_i, L) \end{array}}{\text{NotParallelizable}(L)} \quad [\textsc{LoopMapAccessNotDistinct}]$$

$$\frac{\begin{array}{c} \begin{array}{ll} \text{StatementInLoop}(S_i, L) & [x_i[y_i] := z_i]_i \vee [z_i := x_i[y_i]] \\ \text{StatementInLoop}(S_j, L) & [x_j[y_j] := z_j]_j \vee [z_j := x_j[y_j]] \\ \text{HasStoresInLoop(X[Y], L)} & X \in (\text{pts}(x_i) \cap \text{pts}(x_j)) \end{array} \\ Y \in (\text{pts}(y_i) \cap \text{pts}(y_j)) \\ |\text{Tag}[x_i] \cup \text{Tag}[x_j]| > 1 \vee \text{NotDistinct}(x_i, L) \vee \text{NotDistinct}(x_j, L) \\ |\text{Tag}[y_i] \cup \text{Tag}[y_j]| > 1 \vee \text{NotDistinct}(y_i, L) \vee \text{NotDistinct}(y_j, L) \end{array}}{\text{NotParallelizable}(L)} \quad [\textsc{LoopMapAccessTagConflict}]$$

Figure A.10: Rules for loop parallelization.

A.5 Loop Parallelization (Iteration Non-Aliasing)

We can finally state the conditions under which a loop's heap accesses are parallelizable. We consider the *effects* of each statement S_i: a statement can have read- and write-effects to a (heap abstraction, field) tuple and/or a (map heap abstraction, key abstraction) tuple. Each effect is annotated with the *distinctness* of the associated pointers with respect to loops in context, and the *tags* on these pointers provided by the tag analysis.

Intuitively, a loop is parallelizable (at least with respect to heap accesses) if there is no heap object field or map value slot accessed by more than one iteration unless all accesses are reads. In other words, there can be no RAW, WAW or WAR dependencies on heap locations between loop iterations. (A parallelizable loop must also have no local-variable dependencies across iterations, but this is trivial to check in SSA: any such dependency must flow through a ϕ-node in the loop header.)

This parallelizability condition can be satisfied with the help of distinctness analysis as follows. We ensure that for every accessed (heap abstraction, field)-tuple with at least one write:

- The pointer to every load and store to this field on this abstraction is distinct with respect to the loop

L, and

- The pointer to every load and store to this field on this abstraction has the same tag t.

Intuitively, the first condition ensures that a given statement's accesses do not alias across iterations, and the second condition ensures that two different (static) statements' accesses do not alias across iterations. Without this second condition, distinctness of each pointer alone does not suffice: two different distinct pointers might traverse the same sequence of distinct values in different orders.

Similarly, for map value slot accesses with at least one write:

- The map *or* key pointer on every access is distinct with respect to L, and
- The distinct pointers (maps or keys) have identical tags.

We prove one final lemma that will assist our main result:

Lemma 10. *If N variables $v_1, \ldots, v_N$, defined in loop L, are distinct with respect to L and all have a tag set $Tag[v_k] = \{t\}$, then no dynamic instance of v_k in iteration i can alias any dynamic instance v_l in iteration j of a given instance of L: the set of variable values in each iteration is disjoint with every other iteration.*

Proof. First, by definition of variable distinctness, the assertion holds when $k = l$: a distinct variable cannot alias itself across iterations of an instance of L. Next, consider $k \neq l$. Let us assume that some instance of variable v_k in iteration i and some instance of v_l in iteration j alias; we will show a contradiction.

Both variables have tag t in their tag sets. Either one variable's definition is the tag-originator statement, or a third statement is.

Let us take the first case: without loss of generality, assume that v_k is the tag originator. Then every instance of v_l in iteration i must alias the latest dynamic instance of v_k, and this latest originator instance must be in the current iteration because otherwise it would not dominate the use-site. Now let us consider the cross-iteration aliasing pair above: the dynamic instance of v_l in iteration j aliases an instance of v_k in iteration j, but also (by the assumption above) an instance of v_k in iteration i. But v_k is distinct, so this cross-iteration aliasing cannot occur. Thus, there is a contradiction, and we conclude that v_k and v_l cannot alias across iterations in this case.

In the second case, the separate tag-originator statement must be in the current iteration by the same argument, defining variable v_t, and each dynamic instance of v_k and v_l in iteration i must alias some dynamic instance of v_t in iteration i. This variable v_t must also be distinct w.r.t. L, or else v_l and v_k would not be (by assignment rules for distinctness propagation). Then, similar to the first case above, v_l in iteration j aliases an instance of v_t in iteration j, and v_l in iteration i aliases an instance of v_t in iteration i, so our assumption implies that an instance of v_t in iteration i aliases an instance of v_t in iteration j. This is a contradiction, so this cross-iteration aliasing between v_k and v_l cannot occur. □

We are now able to formalize our loop parallelizability condition with the rules in Fig. A.10. We simplify somewhat by considering only direct heap effects of field load/stores and map load/stores; in actuality, several other IR operations on maps create side-effects on virtual fields, and commutative-store side-effects are possible as well. Also, we rely on an opaque predicate $\text{StatementInLoop}(S_i, L)$ that indicates whether a statement can occur inside (directly or via method calls) the body of loop L. In actuality, these rules are split into "stages" of increasing computational cost so that we first look for conflicts in the direct loop body, then traverse the callgraph to find conflicts in other methods.

We now state our main theorem:

Theorem 1. *In any execution T of program P, all instances of loop L have non-conflicting memory accesses if the four inference rules in Fig. A.10 do not produce a NotParallelizable(L) judgment.*

Proof. Follows directly from Lemma 10. For any given accessed location with at least one write, the above rules ensure that all pointers used to access this location are distinct and have the same tag. By the lemma, no accesses across iterations may alias. In the case of a map access, heap location is determined by the combination of map object and key object; hence, ensuring that either of them does not alias is sufficient to ensure iteration independence. The latter two rules thus require that either tags mismatch or distinctness judgments are missing for *both* the map and key accessed in order to determine a conflict has occurred. □

A.6 Analysis Termination

Above, we have proven the soundness of DAEDALUS and the loop-parallelization rules based upon it. We now argue briefly that the analysis will terminate. The inference rules presented here are *monotonic*. This is because they satisfy the requirements of *stratified negation*: the rules can be split into strongly-connected components (where graph edges are dependencies), or strata, of co-recursive rules. Within such an SCC, no negation of other judgments (Datalog relations) produced within that SCC can occur. This ensures convergence because as judgments are produced, they cannot cause another judgment within that stratum to be withdrawn; the rules in that stratum can thus be iterated until no new judgments are produced, and strata can be visited in dependence order.

Note that within our rules as written here, we have used both *Distinct* and *NotDistinct* judgments for convenience. In implementation, however, only the negative forms (*NotDistinct*, *FieldNotDistinct*, *MapNot-Distinct*, *MapNotDistinctWithinMap*) exist, and the various positive judgments written here are produced as exceptions in other rules that would have produced a negative judgment otherwise.

Appendix B

Definitions and Proofs for Icarus

In this appendix, we provide additional details describing the inference rules of Icarus as well as a soundness proof for the analysis.

B.1 First Pass: Possible Distinctness

As we described earlier in §5.3.1, the first pass of the Icarus analysis computes *possible distinctness.* This analysis determines when it is possible to dynamically prove a value to be distinct given the locations where checks can occur. Because the hybrid static-dynamic system can make a number of static inferences that are dependent on a single successful dynamic check, the result of this analysis contains more possible-distinctness facts than just the set of possible checks.

The possible-distinctness inference rules are derived directly from the original Daedalus analysis' rules, and because they are derived mechanically, we will not reproduce them all here. The transform that produces the new inference rules simply replaces the NotDistinct(v, L) facts in the original rules with NotPossibleDistinct(v, L), and likewise for object fields and map values.

The sole differences to the original static analysis rules have to do with the dynamic checks themselves. We constrain the default rule that produces NotDistinct facts for program variables not covered by other parts of the analysis. In particular, this rule excludes all variables for which a dynamic check is possible. This input to the analysis can be arbitrary, depending on the types of checks that are supported by a particular transform. In our case the analysis considers it possible to check the distinctness of any IR variable for which it knows the corresponding JVM local variable number or JVM operand stack location. Essentially all SSA variables in the Soot-produced IR qualify for checks; only local variables in semantic models do not, because they do not correspond to real code.

To begin our soundness proof, we assert the following lemma:

Lemma 11. *Given a program variable v for which $\neg NotPossibleDistinct(v, L)$, either $\neg NotDistinct(v, L)$, or else there are dynamic check(s) that can be inserted into the program to prove it distinct at runtime.*

Proof. Follows from the design of dynamic-distinctness propagation mechanisms and their correspondence to distinctness inference rules.

For variables with possible direct checks, the lemma is clearly true. We thus consider variables for which no direct check is true, but for which the analysis still claims possible-distinctness. This could arise via (i) assignment from other possibly-distinct variable(s), (ii) a load from a possibly-distinct field via a possibly-distinct pointer, (iii) a load from a map with possibly-globally-distinct values per key and a possibly-distinct key, (iv) or a load from a constant map with possibly-within-map-distinct values per key and a possibly-distinct key. In each of the heap cases, not-distinct bits propagate distinctness through the heap, and these bits can be updated as long as corresponding stores receive possibly-distinct values. In the assignment case, either (Case I) both the assignment sources' definition point and the point of assignment lie within the loop in question, in which case the original variable can be dynamically checked and the loop-iteration serialization behavior from the assignment-source's check have the same effect as a check on the assignment result, or (Case II) either the source or destination lie outside the loop, in which case no possible-distinctness relative to this loop will be propagated. □

B.2 Loop Parallelization Rules

Next, the analysis evaluates a variant of the loop-parallelization rules that operate using possible-distinctness rather than static distinctness. These rules assert needed-distinctness facts which later determine where dynamic checks will occur.

The loop parallelization rules in Icarus are nearly identical to those in Daedalus, except that (i) parallelization-inhibiting conflicts require the conjunction of NotDistinct and NotPossibleDistinct, and (ii) an additional step determines which distinctness facts were actually used after parallelizable loops are chosen. The former change is trivial; the latter change we describe here.

In order to determine which parallelization facts are actually used, we first collect all pointers used to access written-to heap abstractions within the loop body, as before. Each of these pointers in a parallelized loop must be distinct with respect to that loop. Thus, for all loops in the final parallelizable set, we determine which of these pointers is *not* statically distinct. The rules then assert needed-distinctness on these pointers.

In addition, the original static parallelization rules require that all pointers accessing a given heap abstraction *must alias* according to the must-alias analysis. (If this were not the case, they might refer to different sequences of the same distinct objects, thus aliasing across iterations.) This is the case if the must-alias analysis assigns the same tag to all pointers. The rules in the ICARUS system can handle this situation dynamically: for each set of pointers with conflicting tags, the rules assert needed-distinctness on these pointers, with the *same* dynamic-check identifier. To the runtime, these checks appear to be instances of the same check. The checks will thus serialize the loop if any of these accesses is not backward-looking distinct. Note that this is *not* the same as showing that all must alias, but the distinctness (among all potentially-aliasing pointers) is all that is actually required for soundness.

We show one key property now: the analysis will only assert needed-distinctness for variables that are possibly-distinct.

Lemma 12. *Parallelization rules in* ICARUS *will only assert NeedDistinct*(v, L) *if* $\neg$*NotPossibleDistinct*(v, L).

Proof. Follows directly from the rules. A loop will only be parallelizable if all pointers by which it writes to memory are distinct or possibly-distinct. The needed-distinctness logic has identical conditions to specify which pointers must be distinct. Hence, needed-distinctness, which is not asserted when a variable is statically distinct, will only by asserted for pointer variables that are possibly-distinct. □

We argue that these rules choose loops that are soundly parallelizable given our execution strategy in Theorem 2 below.

B.3 Needed Distinctness

After the loop parallelization analysis asserts needed-distinctness facts, ICARUS propagates this "need" backward through inference rules until it eventually arrives at a subset of the possibly-checked locations, inserting checks into the program at those points. The rules that perform this computation are also derived mechanically from the original static rules. However, the derivation is slightly more complex. The rules are thus given explicitly in §5.3.3 as Rule 31 through Rule 38.

In order to show that these rules are sound, we will prove several general properties. First, we show that if a variable is possibly-distinct, and the loop parallelization rules assert needed-distinctness, then the inserted checks will provide *check coverage* for the given variable. What we mean by this is that the inserted check(s) will catch any non-distinct value in the program that would ultimately cause the needed-distinct variable to not be distinct. In the trivial case, the check is directly on the needed-distinct variable. However, checks may also occur long before the needed-distinct variable exists. For the definition of check coverage here, we only consider the successful-check case. The check-failure case is sound, too; we argue this in the next section.

Second, we show that the rules will only assert needed-distinctness for variables that are possibly-distinct. This is an important invariant that begins with the client analysis, loop parallelization, and is preserved during the backward propagation of needed-distinctness by the rules in this section. The invariant enables soundness because it allows the needed-distinctness propagation rules to assume the conditions required for possible-distinctness have been met. Otherwise, impossible situations may arise where there is no possible combination of checks to satisfy needed-distinctness.

We first show that the assignment rules satisfy these properties:

Lemma 13. *If an assignment-result variable* $[v := \phi(v_1, \ldots, v_k)]$ *has NeedDistinct*(v, L)*, then the needed-distinctness inference rules will insert a set of dynamic checks to ensure that* v *is dynamically checked for distinctness.*

Proof. If v has $\neg$NotDistinct(v, L) (is statically distinct), then we are done, trivially: a statically-distinct value is also always dynamically distinct.

Otherwise, if no input v_i has NotPossibleDistinct(v_i, L), it must be that $\neg$AssignInputNotPossible(v, L) (by Rule 31). Then for every input, either $\neg$NotDistinct(v, L) (it is statically distinct), or Rule 32 will create NeedDistinct(v_i, L) facts, propagating the need backward. By the lemmas in this section, this propagated need will eventually result in the required dynamic checks.

Otherwise, if AssignInputNotPossible(v, L), then Rule 33 will insert a local dynamic check. □

Lemma 14. *If assignment-related inference rules assert NeedDistinct*(v, L)*, then it must be the case that* $\neg$*NotPossibleDistinct*(v, L) *or else* $\neg$*NotDistinct*(v, L)*.*

Proof. Follows directly from the rules. Rule 32 will only create a NeedDistinct(v, L) fact if every assignment input has $\neg$NotPossibleDistinct(v_i, L): otherwise AssignInputNotPossible(v, L) would prevent this rule from applying. □

Next, we demonstrate the same for the rules handling field loads (Rule 34 through Rule 36):

Lemma 15. *If a load-result variable* $[x := y.f]$ *has NeedDistinct*(v, L)*, then the needed-distinctness inference rules will insert a set of dynamic checks to ensure that* v *is dynamically checked for distinctness.*

Proof. Two cases exist. First, if either LoadMustCheckLocally$_1(S_i)$ or LoadMustCheckLocally$_2(S_i)$ is asserted for load statement S_i, then the rules insert a local variable-distinctness check on the result of the load, so we are done. Otherwise, if neither of these judgments exists, then the rules assert FieldNeedDistinct$(A.f)$ for every abstraction $A \in$ pts(y) as well as NeedDistinct(y, L). The former ensures that the field's not-distinct bit is properly updated (by Lemma 22 below), and the latter ensures that the pointer is also dynamically checked. □

Lemma 16. *If load-related inference rules assert NeedDistinct*(v, L)*, then it must be the case that either* $\neg$*NotPossibleDistinct*(v, L) *or* $\neg$*NotDistinct*(v, L)*. Likewise, if they assert NeedFieldDistinct*$(A.f)$*, then it must be the case that* $\neg$*FieldNotPossibleDistinct*$(A.f)$ *or* $\neg$*FieldNotDistinct*$(A.f)$*.*

Proof. If needed-distinctness was asserted, then we may assume possible-distinctness was asserted, by a combination of the lemmas in this section and Lemma 12 above. We show this in two cases. First, consider the case where either LoadMustCheckLocally$_1(S_i)$ or LoadMustCheckLocally$_2(S_i)$ holds for the load statement S_i. Then the rules insert a local check on the result variable. For $\neg$NotPossibleDistinct(x, L) to be true, it must have been the case that a local check is possible, so this is sound. Otherwise, if neither of the local-check predicates is asserted, then it must be that $\neg$FieldNotDistinct$(A.f)$ or $\neg$FieldNotPossibleDistinct$(A.f)$ for every $A \in \text{pts}(y)$, because Rule 34 did not apply, and either $\neg$NotDistinct(y, L) or $\neg$NotPossibleDistinct(y, L), because Rule 35 did not apply. So needed-distinctness is only propagated to possible-distinct fields and variables. $\square$

Finally, we consider stores.

Lemma 17. *If a store* $[x.f := y]$ *writes to field* $A.f$*, and FieldNeedDistinct*$(A.f)$ *and FieldNotDistinct*$(A.f)$*, then the needed-distinctness inference rules will insert a set of dynamic checks to ensure that the not-distinct bit on* $A.f$ *is updated.*

Proof. This follows directly from Rule 38. The first set of predicates (in the left-hand column of antecedents) encodes the above conditions. The right-hand column requires some explanation. The antecedents NotConstant(x, L) and NotDistinct(y, L) together find loops for which (i) the stored-to pointer is not constant, meaning that the stored value must be distinct for the field to be distinct, and (i) the stored value is not statically shown to be distinct. If there is no such loop, then the field is already statically distinct, however; because we are considering the case where NotFieldDistinct$(A.f)$, there must exist at least one such loop in the store statement's loop context. Then, for each such loop, we need to dynamically prove distinctness of the stored value with respect to this loop. The rule propagates needed-distinctness to the stored-value variable for all such loops, ensuring the appropriate dynamic checks. $\square$

Lemma 18. *If store-related inference rules assert NeedDistinct*(y, L)*, then it must be the case that either* $\neg$*NotPossibleDistinct*(y, L) *or* $\neg$*NotDistinct*(y, L)*.*

Proof. Follows from possible-distinct rules for stores. A field is only possibly-distinct if for all stores that store to this field, in all loops in context, either the pointer is constant, the stored value is statically distinct, or the stored value is possibly dynamically distinct. Rule 38 only asserts needed-distinctness when the pointer is not constant; hence, the stored value must be statically distinct or possibly dynamically distinct. □

B.4 Dynamic Check Mechanisms at Runtime

We have shown so far that the Icarus analysis will properly insert dynamic checks such that if all checks are successful, the loop-parallelization transform is sound. However, this is not yet sufficient: the reason that we have *dynamic* checks at all is that the checks might fail. (Or, at least, the analysis cannot prove beforehand that they will not fail.) Furthermore, we adopt a non-speculative approach to running the parallelized program, meaning that we cannot roll back the in-progress execution of a loop once we allow it to begin in parallel. We need to ensure soundness in the presence of check failures by some other means.

In this section, we will discuss how dynamic checks are executed at runtime, what synchronization they impose, and how this synchronization results in correct execution in all cases.

B.4.1 Summary of Operation

Here, we briefly summarize the system's operation before proving its soundness. First, the runtime system that parallelizes the loop must keep some state with each loop instance. In particular, for each variable that is checked, the runtime keeps some check-specific state identified by a unique identifier. This state includes a hash-table that records the iteration number in which each value was produced. Each time a check occurs, the runtime probes the hash-table with the value assigned to the variable. If it is present, the runtime fetches the old iteration number. In all cases, it updates the iteration number to the current iteration.

If the iteration number is less than the current one, this is a backward-looking distinctness violation. There are two types of violations. The first is relative to the entire loop instance, since the first iteration, and the second is relative to the set of loop iterations since the last loop serialization.

In either case, the runtime marks the current iteration as *tainted* for the purposes of updating field not-distinct bits. In the latter case (violation since last serialization), the runtime immediately synchronizes on all older iterations, waiting for them to complete before returning to user code and continuing execution of this iteration.

When a dynamically-checking store occurs, it propagates the taint bit from this iteration to the not-distinct bit of every field that it writes. When a dynamically-checking load occurs, it reads the not-distinct bit as well. If the bit is set, it performs a full loop serialization, as a variable distinctness check does, and taints the iteration as well. In addition, it forces all younger iterations to wait for the current iteration to complete. (This is a consequence of one-way field distinctness, as we explain below.)

Finally, any particular instance of a dynamic check must wait for all instances of the same check in older iterations to complete first. In other words, a particular dynamic check must complete in *iteration order*. (This is looser than *program order*, because checks within an iteration may be reordered, for example when multiple checks in an iteration are in a nested parallelized loop.) This synchronization requirement is important to ensure soundness, as we describe below.

B.4.2 Soundness

We first show soundness of *variable distinctness* checks. Recall that ICARUS checks *backward-looking variable distinctness* as defined in Definition 7 in §5.4.1. This property applies to values assigned to single dynamic instances of a variable in the program, and asserts that this value does not alias any past value in a different iteration of the same loop instance.

We first must reason about the state that checks keep and the order in which the state is updated:

Lemma 19. *When a variable-distinctness check probes the hash-table, the fetched value is the last iteration number to produce a given value in original (sequential) program order.*

Proof. This follows from the check serialization: no check will proceed to access the hash-table until all instances of the same check in previous iterations have completed. Hence, checks occur in original program order, so the distinctness facts implied by the check are correct with respect to sequential execution. □

We then establish the main guarantee provided by a variable-distinctness dynamic check with the following lemma:

Lemma 20. *A dynamically-checked variable value is backward-looking distinct with respect to currently in-flight loop iterations once the dynamic check returns.*

Proof. The variable-distinctness check chooses to serialize on older iterations of the loop whenever the value it is checking aliases with a value in some other iteration in flight. To see this, it is sufficient to see that (i) any iteration that is newer than the last-serialization point could still be in flight, precisely because the current iteration has not blocked on its completion; and (ii) the dynamic check chooses to serialize whenever the last iteration to produce this same value is newer than the last-serialization point. By serializing the loop (waiting for older iterations to complete), the check ensures that no other aliasing values are in flight when the check returns. Hence, the value is backward-looking distinct with respect to in-flight iterations at that point, as required. □

Lemma 21. *A dynamically-checked variable value is backward-looking distinct with respect to* all *loop iterations in the instance,* unless *the taint bit is set on the current iteration.*

Proof. This follows directly from the logic that sets the taint bit: it is set exactly when the last-producing iteration is a prior iteration, irrespective of the last-serialization point. □

Given the lemma above, we can now reason about how stores update not-distinct bits. Recall that to extend dynamic checks to field distinctness, we slightly redefined field distinctness in a way analogous to backward-looking distinctness above. In particular, a field is *one-way distinct* if it either does not alias with the same field on any other object in the same heap abstraction, or else does alias and *at least one of* this *or* the other aliasing field has the not-distinct bit set. We thus argue:

Lemma 22. *If a field is dynamically distinctness-checked, every store to that field will correctly set the not-distinct bit to maintain one-way distinctness.*

Proof. First, we establish that there will be at most one store to the field that can affect the one-way distinctness property. This is ensured by Rule 29, which propagates a not-possible-distinct fact for any abstraction field under overlapping stores.

Now, this one store can store an aliasing value to the field only by storing a non-distinct value to different locations. Rule 38 ensures that for every loop in context of the store (capturing all repetition, as per Lemma 2), either the pointer is constant or the value is statically or dynamically distinct. The first two conditions ensure the needed property statically. The last one, dynamic distinctness, suffices as well, as we now argue, despite differing from static distinctness in two ways.

First, a dynamically distinct value is only *backward-looking* distinct. This nevertheless suffices when checking for one-way field distinctness, because for any aliasing pair of values in the sequence of stored values, at least one of the values must be backward-looking non-distinct.

Second, a dynamically distinct value may be non-distinct with respect to the whole loop iteration if the loop iteration taint bit is set. The store dynamic check handles this case by simply propagating the taint bit into the not-distinct bit in the stored field value. □

Finally, we consider loads:

Lemma 23. *Given a field with dynamic one-way field distinctness, a load that performs dynamic checks will produce a dynamically backward-looking distinct value once the check unblocks.*

Proof. The load could produce a dynamically non-distinct value in two ways: by loading from the same field twice, or by loading the same value from two different fields. The former is prevented by ensuring that the pointer is dynamically distinct. The load result will thus have the same distinctness guarantees as the pointer if the field is distinct: namely, it will always be backward-looking distinct with respect to in-flight iterations, and it will be backward-looking distinct with respect to all iterations unless the iteration taint bit is set.

The latter case, loading the same value from two different fields, is handled by checking the not-distinct bit. Because the field is one-way distinct, if the same value is stored twice, one or the other of the aliasing pair must have the not-distinct bit set. Whichever iteration finds the not-distinct bit set performs a serialization in both directions: it waits for all older iterations to complete, and all younger iterations wait for this iteration to complete. Because checks occur in original program order, this ensures that when the field values alias, their respective iterations proceed one at a time, in original program order, satisfying backward-looking distinctness. □

The careful reader will note that the above-described behavior of a load result is slightly different from a variable-distinctness check result in one small way: when values alias, the loop iteration taint bit may be set in *either* the first *or* the second iteration of the aliasing pair. However, this difference does not affect any of our arguments: the iteration taint is only used to set field not-distinct bits, and the one-way distinctness definition tolerates either bit being set.

Given all of the above, we can summarize the guarantee of dynamic distinctness checks with the following lemma:

Lemma 24. *If a program variable is either statically distinct or possibly-distinct, with an asserted needed-distinct fact and corresponding checks, the variable is backward-looking distinct when execution reaches its definition point.*

Proof. We have established proper behavior of all of the individual dynamic checks, so we simply need to establish here that dynamic checks are performed at all of the required locations.

First, if a program variable is statically distinct, then it is necessarily backward-looking distinct, so we dismiss this case easily.

If a variable is possibly-distinct and there is a needed-distinct fact asserted by the client analysis, the variable is backward-looking distinct as well. We argue this by cases, considering the statement type that produces the variable.

If the variable is produced by an assignment, the needed-distinctness propagation logic for assignments ensures that either a local distinctness check is performed, which satisfies the requirement, or the need is propagated to all inputs, and we recursively invoke this lemma.

If the variable is produced by a load, the needed-distinctness propagation logic again either performs a local check or propagates the need backward. If it propagates, it does so both to the pointer variable and to the field on all abstractions from which the value may be loaded. The required property is ensured first by invoking this lemma recursively on the pointer variable, and second by seeing that the not-distinct bit is properly updated. This occurs because the needed-distinctness fact asserted on the field forces dynamic checks on all stores to the field. □

B.4.3 Parallelized Loop Execution

Finally, putting Lemma 24 to use, we show that loop parallelization is sound in this dynamic system.

Theorem 2. *Loop parallelization using dynamic distinctness is sound as long as the loop body performs writes only to backward-looking-distinct pointers. The above rules choose soundly parallelizable loops by asserting the appropriate needed-distinctness facts.*

Proof. Loop parallelization requires separation of memory writes: if two writes, or a write and a read, access the same memory from different loop iterations, then the loop cannot be parallelized (modulo semantic commutativity). Backward-looking distinctness is sufficient to ensure this separation: if a conflict were to occur, one of the writes would be "backward" with respect to the other, hence backward-looking distinctness would prevent this situation from occurring. The above loop-parallelization rules assert needed-distinctness facts on all variables for which the original DAEDALUS rules required ordinary static distinctness. Hence, by Lemma 24 above, these variables are backward-looking distinct, and so by the above argument, parallel execution of the loop is sound. □

Bibliography

[1] BlazingDB: Distributed GPU SQL engine. `https://blazingdb.com/`. 6

[2] GraphQL: Graph query language. `https://graphql.org/`. 44

[3] Janino: A super-small, super-fast Java compiler. `http://janino-compiler.github.io/janino/`. 41, 72

[4] The Java tutorials: parallelism. `https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html`. 5

[5] JGraphT. `http://jgrapht.org`. 41, 72

[6] JLaTeXMath. `https://github.com/opencollab/jlatexmath`. 41, 72

[7] SpiderMonkey JavaScript engine. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`. 113

[8] V8 JavaScript engine. `https://v8.dev/`. 113

[9] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: a system for large-scale machine learning. *OSDI*, 2016. 4, 44

[10] F. Aleen and N. Clark. Commutativity analysis for software parallelization: letting program transformations see the big picture. *ASPLOS*, 2009. 29, 46, 76

[11] ALI Lab at University of Massachusetts. JOlden benchmarks. `ftp://ftp.cs.umass.edu/pub/osl/benchmarks/jolden.tar.gz`. 40, 72

[12] L. O. Andersen. Program analysis and specialization for the C programming language. PhD Thesis, 1994. 10, 18, 31, 49

[13] K. Anderson, T. Hickey, and P. Norvig. JScheme. http://jscheme.sourceforge.net/jscheme/main.html. 41, 72

[14] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: A language and compiler for algorithmic choice. *PLDI*, 2009. 44, 116

[15] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. Opentuner: An extensible framework for program autotuning. *PACT*, 2014. 17

[16] J. Ansel, M. Pacula, Y. L. Wong, C. Chan, M. Olszewski, U.-M. O'Reilly, and S. Amarasinghe. SiblingRivalry: Online autotuning through local competitions. *CASES*, 2012. 17

[17] T. M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. *ISCA*, 1992. 113

[18] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. *PLDI*, 1989. 8, 17

[19] U. Banerjee. Data dependence in ordinary programs. M.S. thesis, UIUC CS Tech. Report UIUCDCS-R-76-837, 1976. 17

[20] M. Basios, L. Li, F. Wu, L. Kanthan, and E. T. Barr. Darwinian data structure selection. *ESEC/FSE*, 2018. 9, 45

[21] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. *CAV*, 2007. 24, 47

[22] G. L. Bernstein, C. Shah, C. Lemire, Z. Devito, M. Fisher, P. Levis, and P. Hanrahan. Ebb: A DSL for physical simulation on CPUs and GPUs. *ACM Trans. Graph.*, 35(2), May 2016. 4, 44

[23] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. *OOPSLA*, 2006. 40, 72

[24] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *PPoPP*, 1995. 71

[25] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. *OOPSLA*, 2009. 40, 62, 71

[26] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya. Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Softw. Pract. Exper.*, 41(1):23–50, Jan. 2011. 41, 72

[27] M. C. Carlisle. Olden: parallelizing programs with dynamic data structures on distributed-memory machines. PhD thesis, 1996. 40, 72

[28] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. *PPoPP*, 2011. 45

[29] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. *VLDB*, 2008. 4, 44

[30] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. *PLDI*, 2013. 6

[31] M. Chevalier-Boisvert and M. Feeley. Simple and effective type check removal through lazy basic block versioning. *ECOOP*, 2015. 113

[32] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. *ISCA*, 1998. 111

[33] A. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. *SOSP*, 2013. 46, 76

[34] A. Colin and B. Lucia. Chain: tasks and channels for reliable intermittent programs. *OOPSLA*, 2016. 112

[35] D. Costa and A. Andrzejak. Collectionswitch: A framework for efficient and dynamic collection selection. *CGO*, 2018. 9, 45

[36] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. *POPL*, 1978. 8, 17

[37] M. De Wael, S. Marr, J. De Koster, J. B. Sartor, and W. De Meuter. Just-in-time data structures. *Onward!*, 2015. 9, 45

[38] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *OSDI*, 2004. 5

[39] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: A domain specific language for building portable mesh-based PDE solvers. *SC*, 2011. 4, 44

[40] J. Eastep, D. Wingate, and A. Agarwal. Smart data structures: An online machine learning approach to multicore data structures. *ICAC*, 2011. 9, 45

[41] M. Ernst. Static and dynamic analysis: synergy and duality. *WODA*, 2003. 112

[42] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Eng.*, 27(2):99–123, Feb 2001. 25

[43] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. *ICSE*, 2000. 25

[44] P. Falstad and R. Hausen. Circuit simulator. `https://github.com/hausen/circuit-simulator`, based on `http://www.falstad.com/circuit/`. 41, 72

[45] P. Feautrier. Dataflow analysis of array and scalar references. *IJPP*, 1991. 8, 17

[46] R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? a shape analysis for heap-directed pointers in C. *POPL*, 1996. 24, 47

[47] R. Ghiya and L. J. Hendren. Putting pointer analysis to work. *POPL*, 1998. 24

[48] P. Ginsbach, L. Crawford, and M. F. P. O'Boyle. CAnDL: a domain specific language for compiler analysis. *CC*, 2018. 4, 44

[49] G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. *PLDI*, 1991. 17

[50] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L. N. Pouchet. Polly – polyhedral optimization in LLVM. *IMPACT*, 2011. 17

[51] S. Idreos, K. Zoumpatianos, B. Hentschel, M. S. Kester, and D. Guo. The data calculator: Data structure design and cost synthesis from first principles and learned cost models. *SIGMOD*, 2018. 9, 45

[52] D. Jeon, S. Garcia, C. Louie, S. K. Venkata, and M. B. Taylor. Kremlin: like gprof, but for parallelization. *PPoPP*, 2011. 113

[53] N. P. Johnson, J. Fix, T. Oh, S. R. Beard, T. B. Jablin, and D. I. August. A collaborative dependence analysis framework. *CGO*, 2017. 8, 18, 76

[54] M. P. Jones. Jacc: Just another compiler compiler for Java. `http://web.cecs.pdx.edu/~mpj/jacc/`. 41, 72

[55] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. *POPL*, 1982. 24, 47

[56] H. Jordan, B. Scholz, and P. Subotic. Soufflé: On synthesis of program analyzers. *CAV*, 2016. 72

[57] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande. Brainy: Effective selection of data structures. *PLDI*, 2011. 9, 45

[58] S. Kamil, A. Cheung, S. Itzhaky, and A. Solar-Lezama. Verified lifting of stencil computations. *PLDI*, 2016. 46

[59] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, Jul. 1967. 8

[60] M. Kim, H. Kim, and C. K. Luk. SD3: A scalable approach to dynamic data-dependence profiling. *MICRO*, 2010. 113

[61] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. *POPL*, 1981. 23

[62] M. Kulkarni, D. Nguyen, D. Proutzos, X. Sui, and K. Pingali. Exploiting the commutativity lattice. *PLDI*, 2011. 9, 46, 76

[63] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. *PLDI*, 2007. 5, 9, 44, 46

[64] V. Kuncak and M. C. Rinard. An overview of the jahob analysis system: project goals and current status. *IPDPS*, 2006. 47

[65] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. *ASPLOS*, 1991. 17

[66] L. Lamport. The parallel execution of DO loops. *CACM*, 17(2), Feb 1974. 8, 14, 17

[67] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. *PLDI*, 2007. 24

[68] D. J. R. López. DJBDD: Java BDD implementation based on hashmaps. https://github.com/diegojromerolopez/djbdd. 41, 72, 80

[69] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. GraphLab: A new framework for parallel machine learning. *Conf. Uncertainty in Artificial Intelligence*, 2010. 4, 44

[70] B. Lucia and B. Ransford. A simpler, safer programming and execution model for intermittent systems. *PLDI*, 2015. 112

[71] K. Maeng, A. Colin, and B. Lucia. Alpaca: intermittent execution without checkpoints. *OOPSLA*, 2017. 112

[72] K. Maeng and B. Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. *OSDI*, 2018. 112

[73] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *T. Software Eng. and Methodology*, 14(1):1–41, Jan. 2005. 21, 40, 62, 72

[74] I. Mokhtarzada. A simple ray tracer written in Java. `https://github.com/idris/raytracer`. 41, 72

[75] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. *ASPLOS*, 1992. 17

[76] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. *SIGMOD*, 2008. 4, 44

[77] D. A. Padua, D. J. Kuck, and D. H. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Trans. on Comp.*, 29(9), Sep 1980. 23

[78] P. M. Petersen and D. A. Padua. Static and dynamic evaluation of data dependence analysis. *ICS*, 1993. 8, 17

[79] P. Peterson and D. Padua. Dynamic dependence analysis: a novel method for data dependence evaluation. *LCPC*, 1992. 113

[80] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Sci Prog J*, 13(4):227–298, 2005. 4, 44

[81] B. Pottenger and R. Eigenmann. Idiom recognition in the Polaris parallelizing compiler. *SC*, 1995. 70

[82] P. Prabhu, S. Ghosh, Y. Zhang, N. P. Johnson, and D. I. August. Commutative set: A language extension for implicit parallel programming. *PLDI*, 2011. 76

[83] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *SC*, 1991. 17

[84] W. Pugh and D. Wonnacott. Constraint-based array dependence analysis. *ACM Trans. Program. Lang. Syst.*, 20(3), May 1998. 8, 17

[85] M. Püschel, J. M. F. Moura, J. Johnson, M. V. D. Padua, B. Singer, J. Xiong, A. G. F. Franchetti, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proc. IEEE, special issue on "Program Generation, Optimization and Adaptation"*, 93, 2005. 44

[86] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4), July 2012. 4, 44, 46

[87] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *PLDI*, 2013. 46

[88] E. Raman, G. Ottoni, A. Raman, M. J. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. *CGO*, 2008. 14, 24

[89] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August. Decoupled software pipelining with the synchronization array. *PACT*, 2004. 14, 24

[90] L. Rauchwerger and D. Padua. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. *PLDI*, 1995. 113

[91] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. *LICS*, 2002. 25

[92] M. Rinard and P. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM TOPLAS*, 19(6), 1997. 29, 46, 76

[93] B. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. *POPL*, 1988. 15, 23

[94] S. Rus, L. Rauchwerger, and J. Hoeflinger. Hybrid analysis: static & dynamic memory reference analysis. *IJPP*, 31, Aug. 2003. 112

[95] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible architectural support for fine-grain scheduling. *ASPLOS*, 2010. 72

[96] Y. Sato, Y. Inoguchi, and T. Nakamura. Whole program data dependence profiling to unveil parallel regions in the dynamic execution. *IISWC*, 2012. 113

[97] B. Scholz, H. Jordan, P. Subotic, and T. Westmann. On fast large-scale program analysis in datalog. *CC*, 2016. 72

[98] A. Sengupta, S. Biswas, M. Zhang, M. D. Bond, and M. Kulkarni. Hybrid static-dynamic analysis for statically bounded region serializability. *ASPLOS*, 2015. 112

[99] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. *PLDI*, 2009. 9, 45

[100] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: The problem based benchmark suite. *SPAA*, 2012. 40, 72

[101] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. *ISCA*, 1995. 113

[102] M. Sridharan and S. J. Fink. The complexity of Andersen's analysis in practice. *SAS*, 2009. 74

[103] B. Steensgaard. Points-to analysis in almost linear time. *POPL*, 1996. 18

[104] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. *ISCA*, 2000. 113

[105] A. K. Sujeeth, K. J. Lee, K. J. Brown, H. Chafi, M. Wu, A. R. Atreya, K. Olukotun, T. Rompf, and M. Odersky. OptiML: An implicitly parallel domain-specific language for machine learning. *ICML*, 2011. 4, 44

[106] G. Tournavitis. Profile-driven parallelisation of sequential programs. PhD Thesis, University of Edinburgh, 2011. 17, 113

[107] S. Treichler, M. Bauer, and A. Aiken. Language support for dynamic, hierarchical data partitioning. *OOPSLA*, 2013. 119

[108] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Mathematical Society*, s2-42, Jan. 1937. 80

[109] A. Udupa, K. Rajan, and W. Thies. ALTER: exploiting breakable dependences for parallelization. *PLDI*, 2011. 76

[110] H. Vandierendonck, S. Rul, and K. De Boesschere. The Paralax infrastructure: automatic parallelization with a helping hand. *PACT*, 2010. 24

[111] M. Weimer, T. Condie, and R. Ramakrishnan. Machine learning in ScalOps, a higher order cloud computing language. *NIPS Workshop on Parallel and Large-scale Machine Learning*, 2011. 4, 44

[112] R. Wilhelm, M. Sagiv, and T. Reps. Shape analysis. *CC*, 2000. 24, 47

[113] M. E. Wolf and M. S. lam. A data locality optimizing algorithm. *PLDI*, 1991. 17

[114] P. Wu, P. Feautrier, D. Padua, and Z. Sura. Instance-wise points-to analysis for loop-based dependence testing. *ICS*, 2002. 9, 18, 51, 76, 112

[115] P. Wu and D. Padua. Containers on the parallelization of general-purpose Java programs. *Intl. J Parallel Prog.*, 28(589), 2000. 9, 46, 76

[116] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler optimization of scalar value communication between speculative threads. *ASPLOS*, 2002. 111