Exploring Alias Analysis Based Optimizations Missed by the Compiler

By

KHUSHBOO CHITRE
MT17020

Under the supervision of Dr. Piyus Kedia and Dr. Rahul Purandare

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDRAPRASTHA INSTITUTE OF INFORMATION TECHNOLOGY DELHI

NEW DELHI - 110020

April, 2026

Exploring Alias Analysis Based Optimizations Missed by the Compiler

By

KHUSHBOO CHITRE
MT17020

A THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDRAPRASTHA INSTITUTE OF INFORMATION TECHNOLOGY DELHI

NEW DELHI - 110020

April, 2026

Certificate

This is to certify that the thesis titled *Exploring Alias Analysis Based Optimizations Missed by the Compiler* being submitted by *Khushboo Chitre* to the Indraprastha Institute of Information Technology Delhi, for the award of the degree of Doctor of Philosophy, is an original research work carried out by her under my supervision. In my opinion, the thesis has reached the standard fulfilling the requirements of the regulations relating to the degree.
The results contained in this thesis have not been submitted in part or full to any other university or institute for the award of any degree or diploma.

Dr. Piyus Kedia
29 April, 2026
Department of Computer Science & Engineering, IIIT-Delhi

Dr. Rahul Purandare
29 April, 2026
Department of Computer Science & Engineering, IIIT-Delhi
University of Nebraska-Lincoln, USA

Declaration Regarding Use of AI in Thesis

I acknowledge that I am fully responsible for the entire content of my thesis, including any sections assisted by online tools, including Artificial Intelligence-based tools. I accept full accountability for any violations of ethical standards in publications arising from the use of such tools.

Khushboo Chitre
MT17020
29 April, 2026

Abstract

Alias analysis is a technique to determine whether a pair of pointers points to overlapping memory locations. Alias information (especially non-overlapping of pointers) is critical for enabling key transformations such as vectorization, redundant code elimination, loop-invariant code motion, and so on.

Context-sensitive interprocedural alias analyses are generally more precise than intraprocedural analyses, but they are often not scalable for large, real-world programs.While context-insensitive interprocedural analyses are more scalable, they significantly sacrifice precision due to call-site merging and conservative assumptions. As a consequence, most of the production compilers sacrifice precision for scalability and implement intraprocedural alias analysis. Moreover, purely static alias analyses are often too conservative when aliasing depends on runtime inputs, calling contexts, or control-flow conditions, especially in cases where pointers overlap only for specific or rare executions.

We first proposed a tool to estimate the upper bound on the performance of SPEC benchmarks in the presence of the most precise aliasing information. The key idea was to profile SPEC benchmarks to log alias information, use them to optimize the program, and obtain an upper bound on the likely performance improvement using the most precise alias analysis. Here, the upper bound is an empirical, input-dependent estimate of the performance improvement achievable through improved aliasing precision. We found that an execution time improvement of up to 11.56% is possible for the SPEC benchmarks. Additionally, we

found that up to 53.32% execution time improvement is possible for Polybench benchmarks. Polybench benchmarks are used to evaluate the effectiveness of loop transformations.

To improve the precision of alias analysis, several prior works propose combining code versioning with dynamic checks to preserve the scalability of alias analyses while selectively improving precision in performance-critical regions. These approaches do not increase alias analysis precision, instead, they use runtime checks to improve the effectiveness of alias analysis in enabling optimizations. These techniques are particularly effective when applied at the loop granularity, since loops often dominate runtime. However, most of these approaches are restricted to loops with either loop-invariant bounds or have very high overheads of dynamic checks, which limit their applicability. This thesis proposes two approaches to reduce the overhead of dynamic checks to constant time.

The first approach constrains the allocation size and alignment of the memory objects using a segment-based allocator to reduce the overhead of the dynamic checks to constant time. We achieved a CPU performance improvement of up to 1.47% with a geometric mean of 0.55% for the SPEC benchmarks. The allocator introduced a maximum overhead of 8.36%, with a geometric mean of 1.47% across all SPEC benchmarks. For Polybench benchmarks, we achieved up to 51.11% CPU performance improvement. The allocator introduced a maximum overhead of 5.3%, with a geometric mean of 0.21% across all Polybench benchmarks.

Our second approach reduces allocator overhead by selectively constraining memory allocations. We also proposed a region-based allocation strategy that eliminates the need for memory accesses in the dynamic checks. This approach never resulted in performance degradation and achieved improvements of up to 1.88% with a geometric mean of 0.58% on SPEC benchmarks. The maximum

CPU overhead of our allocator is 0.57% with a geometric mean of -0.2% for SPEC benchmarks.

Both our approaches outperform a previous approach to disambiguate pointers, which reported a 29% overhead for a SPEC CPU 2006 benchmark. The primary contribution of this thesis is the development of a dynamic disambiguation approach that can be applied to enhance the performance of real-world, large-scale applications.

To my family.

Acknowledgements

This thesis would not have been possible without the support and guidance of many individuals and institutions.

First and foremost, I express my deepest gratitude to my thesis advisors, Dr. Piyus Kedia and Dr. Rahul Purandare, for their invaluable mentorship, constant encouragement, and unwavering support throughout the course of this work. Their insights, feedback, and patience have been instrumental in shaping my research journey. Working with them taught me how to break down research problems into manageable pieces, how to stay focused while iterating through ideas, and how to communicate technical details effectively. Their constant motivation kept me focused during challenging phases, and their belief in my work gave me the confidence to push my limits.

I am sincerely thankful to the TCS Research Foundation for supporting my Ph.D. through the TCS Research Fellowship Program, which provided me with the resources and freedom to pursue my work. The fellowship offered financial assistance and recognition that further encouraged me to strive for impactful research. I appreciate the continued support from TCS in fostering academic collaboration and creating opportunities for young researchers. I am also grateful to SIGPLAN OOPSLA PAC and ACM-India/IARCS for providing travel grants that enabled me to present my work and interact with the broader research community at conferences.

I would like to thank my thesis examiners, Prof. Anders Møller, Prof. Santosh Nagarkatte and Prof. V Krishna Nandivada, for their time and thoughtful feedback, which helped improve the quality of this thesis. I am also grateful to my annual review committee members, Dr. Vivek Kumar (IIITD), Dr. Sambuddho Chakravarty (IIITD), and Dr. Bapi Chatterjee (IIITD) for their consistent guidance and constructive suggestions over the years. I extend my sincere appreciation to my comprehensive exam committee, Dr. Subhajit Roy (IITK), Dr. Vivek Kumar (IIITD), and Dr. Sambuddho Chakravarty (IIITD) for their insightful feedback and guidance, which played an important role in shaping this thesis.

I would also like to acknowledge all the faculty members at IIITD for creating a vibrant academic environment that continuously inspired me. I am thankful to the IT team and administrative staff at IIITD for their constant support and assistance throughout my doctoral studies.

A special thanks to my wonderful lab mates — Dhriti Khanna, Ridhi Jain, Nikita Mehrotra, Devika Sondhi, Gopi Krishna Menon, Sunil Kumar, and Vanshika Jain for the many technical discussions, shared meals, casual chats, celebrations of small and big achievements, and the constant guidance and learning that made this journey both meaningful and enjoyable. Working alongside them made the lab feel like a second home, turning long hours into memorable experiences. Whether it was brainstorming over code or unwinding with light-hearted conversations, their presence made a lasting difference in my Ph.D. life.

Finally, and most importantly, I am forever indebted to my family. I thank my parents, Madhvi Chitre and Pravir Chitre, for their unconditional love, countless sacrifices, and for always keeping me motivated through every high and low. Their steady belief in my potential gave me the strength to keep going, even during the most challenging phases of this journey.

I am deeply grateful to my husband, Arjun Malhotra, for being my constant source of strength, understanding, encouragement, and emotional support. He has stood by me unconditionally by helping me deal with every failure to celebrating each small success. His thoughtful feedback on my writing and willingness to help me through difficult drafts made this process more manageable.

Last but not the least, I would like to remember my grandparents, Suparna Chitre and Prakash Chitre, for all the values and life lessons they instilled in me. I deeply wish they were here to witness the successful end of this journey. Their memory continues to inspire me every day.

List of Publications

1. **K. Chitre**, P. Kedia and R. Purandare, 2026. "HORIZON: Estimating Alias Analysis Precision Bounds and Their Impact on Performance". 2026 International Conference on Compiler Construction (CC 2026). [Just Accepted]
2. **K. Chitre**, P. Kedia and R. Purandare, 2023. "RAPID: Region-based Pointer Disambiguation". 2023 ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA 2023).
3. **K. Chitre**, P. Kedia and R. Purandare, 2022. "The Road Not Taken: Exploring Alias Analysis Based Optimizations Missed by the Compiler". 2022 ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA 2022).

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we motivate the need for context-sensitive interprocedural alias analysis as a means to improve the precision of alias information used by compilers. We provide a brief overview of current practices for evaluating and enhancing alias analysis precision, highlighting the key challenges that hinder their adoption in real-world compilers. These challenges form the basis for the contributions proposed in this thesis.

1.1 The Need for Precise Alias Analysis in Compiler Optimizations

Compiler optimizations play a vital role in improving the performance of programs by enabling transformations, such as dead store and load elimination [1, 2], loop-invariant code motion [3], code vectorization [4], and so on. These optimizations often rely on precise information about how memory is accessed—particularly whether two pointers may refer to the same memory location. Alias analysis serves this purpose by identifying potential overlaps between pointers. However, when the analysis lacks precision, the compiler must conservatively assume that pointers may alias, which prevents the application of many aggressive optimizations.

To illustrate this, consider the example in Figure 1.1a, a loop that adds elements from two arrays `b` and `c`, storing the result into array `a`. In its scalar form, the loop processes one element at a time. However, with sufficient aliasing information, the compiler can transform this loop into a vectorized version

```
void foo(int *a, int *b,
    int *c, int *size) {
1.  for(int i = 0;
      i < *size; i++)
2.    a[i] = b[i] + c[i];
}
```

(a) The original code.

```
void foo(int *a, int *b,
     int *c, int *size) {
3.   int t = *size;
4.   int i;
5.   for (i = 0; i+3 < t;
       i = i+4)
6.     a[i:i+3]
       = b[i:i+3] + c[i:i+3];
7.
8.   for (; i < t; i++)
9.     a[i] = b[i] + c[i];
10.
}
```

(b) The transformed code using precise alias information.

Figure 1.1: An example demonstrating how precise alias information can enable optimizations. If alias analysis can determine that `a`, `b`, `c`, and `size` do not overlap, the compiler can safely apply loop vectorization, allowing the loop iterations to execute parallely.

that processes multiple elements in parallel, as shown in Figure 1.1b, which might result in significant performance improvements. This transformation is only safe if alias analysis can prove that the memory pointed to by `a` do not overlap with `b`, `c`, and `size`. The correctness of this transformation is defined under serial semantics, and therefore requires that the non-aliasing information be preserved across function boundaries. However, most production compilers, such as LLVM, do not retain such information interprocedurally due to the high cost of maintaining precise alias relationships across calls. As a result, the compiler is forced to make conservative assumptions about pointer aliasing, which can prevent optimizations from being applied. In this example, such conservatism leads to the vectorization opportunity being missed entirely. This case illustrates the power of precise alias information—it not only enables transformations like vectorization but also helps unlock the full optimization potential of modern compilers. Therefore, improving alias analysis precision is crucial for exposing optimization opportunities that would otherwise remain hidden.

1.2 Context-Sensitive Interprocedural Alias Analysis

One way to improve the precision of alias analysis is to perform context-sensitive interprocedural analysis. Alias analysis can be conducted either intraprocedurally, where alias relationships are determined within the boundaries of a single function, or interprocedurally, where the analysis spans across function calls. While intraprocedural analysis is faster and more scalable, it often lacks the information necessary to reason about pointer interactions introduced via function arguments or return values.

In addition to interprocedural reasoning, context sensitivity plays a critical role in enhancing precision. A context-sensitive alias analysis distinguishes between different call sites of a function and maintains aliasing information specific to each calling context. In contrast, context-insensitive analysis merges information from all call sites, leading to conservative assumptions and potential loss of precision. Preserving call site-specific aliasing behavior allows the compiler to make more informed decisions about which memory accesses can safely be optimized.

Consider again the example shown in Figure 1.1a, where the function `foo` takes pointers `a`, `b`, `c`, and `size` as arguments. To determine whether the loop can be safely vectorized, the compiler must know that `a` does not alias with any of `b`, `c`, or `size`. Since these variables are passed as function arguments, the aliasing relationships must be inferred at the call site and preserved across the function boundary. A context-sensitive interprocedural alias analysis can capture this non-aliasing information and make it available inside the function, thereby enabling optimizations such as vectorization.

Over the years, many algorithms have been developed to track and preserve aliasing information across function calls in a context-sensitive manner [5–20]. However, even the most precise context-sensitive interprocedural alias analysis fails to correctly deduce aliasing relationships when multiple allocations are performed at the same allocation site. For example, when `malloc` is called within a loop to allocate multiple elements of an array, the array's elements are allo-

cated using the same allocation site.

```
void foo(int **a, int n) {
  for (int i = 0; i < n; i++)
    a[i] = (int *)malloc(36);
}
```

For a call like bar(a[x], a[y]), where x and y are two different indices of array a, context-sensitive interprocedural alias analysis would fail to accurately assess the aliasing relationship between a[x] and a[y], because both are allocated at the same site. The default may-alias relationship would be assumed, even though the actual relationship should be no-alias.

Context-sensitivity also fails to precisely assess aliasing relationships that are dependent on user-input. So, even though context-sensitive algorithms are more precise than context-insensitive analyses, their precision is limited due to their static nature. Additionally, they are computationally expensive and require a significant amount of memory to store aliasing information for each calling context. Due to these scalability and precision concerns, production compilers such as LLVM do not adopt such techniques, opting instead for more scalable intraprocedural analyses.

Compiler developers prioritize scalability over precision, leading to missed optimization opportunities in real-world applications. In both examples presented, the lack of precise aliasing information causes the compiler to assume that the arguments may alias, thereby blocking loop vectorization and similar optimizations. This highlights a fundamental limitation in existing compiler infrastructures and motivates the need for alternatives that can retain precision without compromising the scalability of the algorithms. Another problem with the static alias analysis algorithms is that they require pointers to be either an alias or not an alias 100% of the time. For example, even if two pointers overlap 0.01% at runtime, the static analysis algorithm will consider them as potentially overlapping accesses, and will not perform any optimization that would have worked perfectly 99.99% of the time. Such cases can only be handled using dynamic checks. The primary focus of this thesis is to propose lightweight

dynamic checks to improve the performance of code regions that work with non-overlapping pointers most of the time, as the non-overlapping property is key for many prominent compiler optimizations.

1.3 Identified Problems and Proposed Solutions

One of the key challenges in alias analysis is understanding whether increasing its precision leads to meaningful performance improvements. While several alias analysis algorithms offer higher precision, their practical adoption is often hindered by concerns about scalability and unclear payoff. Without concrete evidence of optimization benefits, compiler developers tend to favor faster, less precise analyses.

A common approach to estimating the potential of improved alias analysis is to make a static assumption that none of the pointers alias [21, 22]. This provides a loose upper bound on the number of optimizations that could be applied under perfect alias information. While useful as a theoretical estimate, this method is overly optimistic and does not reflect actual runtime behavior or achievable gains.

To go beyond static approximations, we propose a profiling-based tool (Chapter 4) that dynamically identifies the actual alias relationships present during program execution. The tool operates in two steps: first, it instruments the program to detect real alias interactions at runtime; second, it uses this information to re-enable optimization opportunities previously missed due to conservative alias assumptions. This approach offers a more accurate estimation of the practical benefits of improved alias information.

Our tool is designed to identify optimization opportunities that, if enabled, can lead to measurable performance improvements. Revealing which missed optimizations are impactful allows developers to focus on cases where improving alias analysis precision would translate directly into runtime gains. Furthermore, it highlights the most influential alias relationships that block performance-critical transformations, helping guide targeted enhancements to the alias analysis without incurring unnecessary overhead.

Another class of approaches addresses alias imprecision by disambiguating pointers, i.e., determining at runtime whether two pointers may refer to overlapping memory locations, at runtime using lightweight techniques [23–25]. These approaches typically combine code versioning with statically generated dynamic checks to determine, at runtime, whether an optimized version of the code can be safely executed. Since loops are often the most performance-critical sections in programs, such techniques are commonly applied at the granularity of loops. Notably, our profiling-based tool supports this design choice by showing that loop-level optimizations can offer substantial performance benefits when aliasing restrictions are relaxed.

Many production compilers, including LLVM and GCC, implement loop versioning with dynamic checks [26, 27]. These techniques are scalable and practical for real-world use. Still, they are limited to a subset of loops—specifically, those with loop-invariant bounds, where the memory access patterns are predictable at compile time. Although a dynamic approach supports more general loops [24], it relies on metadata lookups to identify object boundaries, which incurs a runtime cost of $O(\log n)$ per dynamic check. This overhead makes the approach unsuitable for large-scale applications.

To make runtime disambiguation viable for a broader class of loops and applications, we identified the need to reduce the overhead of dynamic checks. Our solution is to constrain memory allocations (restricting the size and alignment of memory objects at allocation time) in a way that makes pointer disambiguation significantly cheaper, discussed in Chapter 5. Specifically, we introduce an approach that constrains the size and alignment of memory objects during allocation, enabling pointer disambiguation using just a single memory access, thereby reducing the check overhead to $O(1)$. This is achieved by implementing a segment-based allocator that partitions memory into aligned segments for efficient disambiguation.

While this approach provides fast checks, it constrains all memory allocations, which leads to additional CPU and memory overhead, limiting its applicability to some real-world programs. To improve scalability, we propose a second

technique (Chapter 6) that constrains only those memory allocations that are potentially accessed within additionally versioned loops. We implement a profiler to identify such memory allocations, reducing unnecessary allocator overhead. Furthermore, to eliminate the memory access required for dynamic checks in our first approach, we design a region-based allocator that assigns the memory allocations to different memory regions that require the dynamic checks. Our second scheme could improve the performance of most of the SPEC CPU 2017 benchmarks.

Our earlier profiling-based tool plays a key role in this design by identifying precise alias relationships that require dynamic disambiguation at runtime. By combining selective memory constraints with profiling insights, we enable scalable, low-overhead disambiguation, unlocking precise loop-level optimizations in real-world applications.

Although our second scheme performs better than the first scheme, it has a profiling phase. Our first approach can enhance the performance of large benchmarks without requiring a profiler.

1.4 Thesis statement

This dissertation confirms the following thesis statement:

Thesis Statement

Runtime pointer disambiguation through constrained memory allocation enables scalable and aggressive compiler optimizations that rely on precise alias information.

More precisely,

TS - 1 The logarithmic time complexity of pointer disambiguation can be improved to constant time (with one memory access) by constraining the size and alignment of all memory allocations using a segment-based allocator.

TS - 2 Pointer disambiguation can be made even more efficient by constraining only those memory allocations that may be accessed in additionally versioned loops. Additionally, using region-based allocation instead of segment-based allo-

cation can reduce the CPU and memory overheads associated with constrained allocation.

1.5 Contributions

This thesis explores practical approaches for evaluating and improving alias analysis precision in compilers while maintaining scalability. Since aggressive optimizations rely on precise aliasing information, we aim to make such precision useful and affordable in real-world applications. This work makes the following key contributions to bridge that gap:

- We present a profiling-based tool (Chapter 4), integrated with LLVM, that helps estimate the potential benefits of improving alias analysis precision. It dynamically identifies actual alias relationships observed during execution and uses this information to highlight missed optimization opportunities. It allows compiler developers to assess whether refining alias analysis precision will likely yield measurable performance gains.
- We propose a runtime disambiguation technique (Chapter 5), integrated with LLVM, that uses a segment-based memory allocator to constrain the size and alignment of memory objects. This design enables constant-time ($O(1)$) pointer disambiguation through a single memory access, making it feasible to enable optimizations such as loop vectorization in contexts where static alias analysis is too conservative.
- To reduce the memory and CPU overheads introduced by globally constraining all allocations, we introduce a second approach (Chapter 6), implemented in LLVM, that selectively applies memory constraints only to allocations that may be accessed within additionally versioned loops. A profiler is used to identify such allocations, and a region-based allocator is implemented to eliminate memory access overhead during dynamic checks.
- Together, these contributions enable a practical balance between precision and scalability in alias analysis. By combining dynamic profiling with lightweight runtime disambiguation, we can unlock previously missed opti-

mizations in real-world applications without the overheads of full context-sensitive interprocedural analysis.

1.6 Extension of our profiling-based tool beyond the M.Tech Thesis

This thesis revisits and significantly expands upon Horizon, a profiling-based tool originally developed during my M.Tech thesis. Horizon served as a diagnostic tool, which introduced a runtime profiler to estimate the potential precision gains of static alias analysis when augmented with missed must-alias information. Its primary goal was to demonstrate how much improvement static analysis could achieve if nearly perfect must-alias information were available.

However, through deeper investigation and practical evaluation, it became clear that many impactful optimizations, particularly vectorization and loop invariant code motion, depend not on must-alias information, but on no-alias guarantees. Recognizing this, we decided to extend Horizon to include profiling-based no-alias detection. In doing so, Horizon evolved from a profiler capturing selective alias information into a profiler capturing complete alias information. Thus, it becomes a practical guide for compiler developers, helping them determine where improving alias precision could yield measurable performance gains. Furthermore, we extended Horizon to provide actionable feedback to users and developers, highlighting specific alias pairs that could potentially inhibit optimizations.

In this thesis, Horizon has been fully rewritten to support these broader objectives. The evolved version of Horizon has been named Horizon+. It now instruments both must-alias and no-alias behaviors, and connects profiling results directly to potential optimization passes. The result is a completely restructured and reimplemented version of Horizon that is central to this thesis. It bridges profiling and optimization by helping estimate the performance potential of improving alias precision and guiding dynamic disambiguation strategies introduced in later chapters.

1.7 Relationship to the Publications

This thesis builds upon and extends several of our previously published papers, which are listed in page viii. The thesis chapters build directly on the conference versions and therefore share substantial common content. Chapter 4 extends the corresponding conference paper by introducing additional material and analysis that were omitted from the publication, while Chapters 5 and 6 are based on the third and second papers listed in page viii, largely follow their respective conference versions with revised introductions and a unified presentation.

Chapter 4 is based on our first conference paper listed in page viii. In the thesis, the tool presented in the paper is referred to as Horizon+, as this work extends an earlier M.Tech thesis, as discussed in Section 1.6. The chapter introduction has been modified to focus specifically on the work discussed in the chapter. In addition, the thesis includes a detailed discussion on providing feedback to the user, corresponding to Research Question 3, which is presented in Section 4.4.4. This material was not included in the conference paper, which primarily focuses on estimating a loose upper bound on optimization potential, identifying useful and harmful optimizations, and enabling the safe insertion of the `restrict` keyword based on inferred non-aliasing information for given inputs. The conference paper also omits the analysis presented in Section 4.4.6, which examines the relationship between performance impact and precision improvement potential.

1.8 Outline of Dissertation

The rest of the dissertation is organized as follows. We present the necessary background and related work on alias analysis in Chapter 2 and Chapter 3, respectively. Chapter 4 introduces our profiling-based tool named, Horizon+ that quantifies the performance benefits of increased alias analysis precision. Chapter 5 and Chapter 6 describe two scalable, lightweight approaches to disambiguate pointers at runtime. The first approach, encapsulated in a tool

called Scout, leverages a segment-based memory allocator to achieve constant-time pointer disambiguation but introduces some allocator overhead. To mitigate this, the second approach introduces a tool named Rapid, which implements a region-based allocator that selectively constrains memory allocations based on profiling data, thereby reducing overhead while preserving optimization potential. Together, these contributions aim to bridge the gap between alias analysis precision and scalability in modern compilers. Finally, Chapter 7 concludes the dissertation and outlines directions for future work.

Chapter 2

Background

2.1 Alias Analysis

Alias analysis determines whether a pair of pointers point to overlapping memory locations, which is essential for enabling safe and effective compiler optimizations.

Type of alias relationships. Generally, there are three types of relationships between a pair of pointers: `must-alias`, `no-alias`, and `may-alias`. The `must-alias` relationship indicates that both pointers always refer to the same memory location at a given program point. The `no-alias` relationship guarantees that the pointers never point to overlapping memory. The `may-alias` relationship signifies that the pointers might refer to overlapping memory locations, but not in all execution paths. Among these, `must-alias` and `no-alias` are particularly valuable, as they provide deterministic information that compilers can leverage to perform aggressive optimizations safely.

Consider the following example:

```
int main() {
  int a, b;
  int x, y;
  int *p = &x;
```

```
  int *q = &x;
  int *r = &y;
  int *t, *s = &a;
  if (...)
    t = &a;
  else
    t = &b;
  return 0;
}
```

In this example,

- `p` and `q` both point to `x`, so they exhibit a `must-alias` relationship.
- `p` and `r` point to different variables (`x` and `y`), so they exhibit a `no-alias` relationship.
- `s` always points to `a`, while `t` may point to either `a` or `b` depending on the branch taken. Thus, `s` and `t` exhibit a `may-alias` relationship.

Intraprocedural and interprocedural. Alias analysis can be conducted either `intraprocedurally` or `interprocedurally`. Intraprocedural alias analysis restricts its scope to a single function, analyzing aliasing within the function body while over-approximating the effects of function calls and external interactions. In contrast, interprocedural alias analysis considers alias relationships across function boundaries as in Andersen's inclusion-based analysis [28]. It tracks how pointer values flow between callers and callees through function parameters, return values, and global variables. While interprocedural analysis tends to be more precise, it is also more computationally expensive and less scalable.

Consider the following example:

```
void foo(int *x, int *y);
```

```
int main() {
  int a, b;
  int *p = &a;
  int *q = &b;
  foo(p, q);
  return 0;
}
```

In this example, pointers `p` and `q` are passed to the function `foo`, and they point to distinct memory locations `a` and `b`, respectively. An intraprocedural alias analysis performed within `foo` does not analyze the calling context from `main`, and thus conservatively assumes that `x` and `y` might alias. As a result, optimizations inside `foo` that rely on knowing whether `x` and `y` are distinct might be inhibited.

On the other hand, an interprocedural alias analysis takes into account the call site in `main` and can determine that `x` and `y` point to different variables. Therefore, it can be safely concluded that the parameters `x` and `y` in `foo` do not alias. This can enable the compiler to apply more aggressive optimizations inside `foo` that would otherwise be blocked under conservative assumptions.

Context-insensitive and context-sensitive. Context-insensitive alias analysis treats all invocations of a function as equivalent; it does not distinguish between different call sites and merges alias information across them, as in Andersen's inclusion-based analysis [28]. On the other hand, context-sensitive alias analysis differentiates between call sites of the same function, maintaining alias information that is specific to each calling context [29]. While context-insensitive analysis is more scalable and faster, context-sensitive analysis offers greater precision at the cost of increased time and memory usage. As a result, most production compilers prioritize compilation speed and scalability, and thus rely mainly

on intraprocedural alias analysis [30, 31], with limited use of context-insensitive interprocedural information.

Consider the following example:

```
void foo(int *a, int *b, int *c, int *size) {
  for(int i = 0; i < *size; i++)
    a[i] = b[i] + c[i];
}

int main() {
  int size = 100;
  int p[size], q[size], r[size];
  foo(p, q, r, size);
  foo(p, p, r, size);
  return 0;
}
```

In this code, `foo` is called twice with different aliasing behaviors among the arguments. In the first call, all three pointers (`a`, `b`, and `c`) point to different arrays, `p`, `q`, and `r` respectively i.e. there is `no-alias` relationship between them. In this case, a compiler can safely apply optimizations such as loop invariant code motion and loop vectorization, as discussed later in Figure 2.4b.

In the second call, both pointers `a` and `b` point to `p`, i.e., they `must-alias`. This introduces a potential dependency between reads from `b[i]` and writes to `a[i]`, preventing the optimizations.

A context-insensitive alias analysis merges alias information across all calls to `foo` and concludes conservatively that `a` and `b` might point to overlapping memory locations. This inhibits optimizations in both calls, even where they would be safe.

By contrast, a context-sensitive alias analysis distinguishes between the two calls. It allows optimizations in the first call by detecting `no-alias`, while preserving correctness in the second by recognizing the `must-alias` relationship. This leads to better overall performance without sacrificing safety.

There are multiple ways in which the context-sensitive optimization can optimize this program. In the first approach, the first call to `foo` can be replaced with `foo_no_alias`. `foo_no_alias` assumes that the pointers don't overlap and thus the compiler can generate an optimized code for `foo_no_alias`. In the second approach, the compiler can selectively inline the first call to `foo` and thus it can be optimized using the non-aliasing information known to `main`. Additionally, if `foo` can also be optimized using must-alias information, the second call to `foo` can be replaced with `foo_alias_2`, which assumes that the first two arguments are aliases.

Flow-insensitive and flow-sensitive. Flow-sensitive alias analysis considers the control flow of the program and tracks how pointer relationships changes at different program points, enabling more precise results [29]. On the other hand, flow-insensitive alias analysis ignores program order and evaluates pointer relationships in a cumulative manner, assuming that all assignments may happen in any order, as in Andersen's inclusion-based analysis [28]. While flow-insensitive analysis is more scalable and easier to implement, flow-sensitive analysis provides better precision, which is crucial for enabling certain optimizations.

A flow-sensitive analysis observes the program step by step. It determines that at line 4, `p` points to `a`, and at line 5, `p` points to `b`, but no longer points to `a`. This precise tracking allows the compiler to reason accurately about memory access at each program point.

In contrast, a flow-insensitive analysis merges all assignments to `p` and concludes that `p` may point to `a` and `b` throughout the function. This over-

approximation limits the ability of the compiler to optimize or verify memory accesses precisely.

Consider the following example:

```
int main() {
  int a, b;
  int *p;
  p = &a; // Line 4
  p = &b; // Line 5
  return 0;
}
```

Field-insensitive and field-sensitive. Field-sensitive alias analysis distinguishes between different fields of a built-in or user-defined data structure. It tracks aliasing information for each field individually. This granularity allows more precise reasoning about memory accesses involving complex data structures. In contrast, field-insensitive analysis treats all fields of a structure as a single abstract location. This means updates or accesses to one field are conservatively assumed to potentially affect all fields, reducing precision but improving scalability and simplifying implementation. Consider the following example:

```
struct {
  int *a;
  int *b;
} p;

int main() {
  int x, y;
  p.a = &x;
  p.b = &y;
  return 0;
}
```

A field-sensitive analysis recognizes that `p.a` points to `x` and `p.b` points to `y`. Since `x` and `y` are distinct, `p.a` and `p.b` do not alias. On the other hand, a field-insensitive analysis treats both `p.a` and `p.b` as accesses to the same abstract location (i.e., `p.*`), and thus may conservatively conclude that they could alias, even though they point to different memory locations. This conservative behavior may block optimizations that rely on knowing fields are distinct.

2.2 Optimizations Leveraging Alias Analysis Information

Many optimizations rely on alias analysis to generate more efficient, yet semantically equivalent, versions of a program. These optimizations depend on `must-alias` and `no-alias` relationships to safely reason about pointer interactions and apply transformations without changing program behavior. This section describes several key optimizations used in our study that utilize alias analysis information.

Global Value Numbering (GVN). GVN [32] eliminates redundant computations by identifying expressions that are provably equivalent. It assigns a unique symbolic number to each variable and expression; if two expressions receive the same number, one is redundant and can be removed. GVN operates at the function level, rather than within individual basic blocks, which is why it is referred to as "global" value numbering. This optimization relies on the program being in Static Single Assignment (SSA) form, where each variable is defined exactly once, simplifying redundancy detection.

Consider the example in Figure 2.1a, which computes the sum of two array elements twice and returns the sum of those results. GVN assigns unique value numbers to expressions with equivalent computation. In this case, both instances of `a[i] + b[i]` are identical and operate on the same memory lo-

```
int foo(int *a,
  int *b, int i) {
1.  int x = a[i] + b[i];
2.  int y = a[i] + b[i];
3.  return x + y;
}
```

(a) The original code.

```
int foo(int *a,
  int *b, int i) {
4.  int x = a[i] + b[i];
5.  return x + x;
}
```

(b) The transformed code using GVN.

Figure 2.1: An example demonstrating how GVN can optimize the code.

cations without any intervening modifications to `a[i]` or `b[i]`. GVN detects this redundancy, assigns the same value number to both expressions, and eliminates the repeated computation on line 2. The transformed code, shown in Figure 2.1b, avoids redundant loads and can improve the performance.

Loop Invariant Code Motion (LICM). LICM [3] improves performance by moving computations that yield the same result on every loop iteration outside the loop body. These operations can include memory loads (moved before the loop) and stores (moved after the loop), reducing the number of times they are executed. LICM also facilitates register allocation for invariant values, avoiding repeated memory accesses across iterations.

```
void foo(int *a, int *b,
    int *c, int *size) {
1.  for(int i = 0;
      i < *size; i++)
2.    a[i] = b[i] + c[i];
}
```

(a) The original code.

```
void foo(int *a, int *b,
     int *c, int *size) {
3.   int t = *size;
4.   for(int i = 0;
       i < t; i++)
5.    a[i] = b[i] + c[i];
}
```

(b) The transformed code.

Figure 2.2: An example demonstrating how LICM can optimize the code.

Consider the example in Figure 2.2a. The code computes the element-wise sum of two arrays `b` and `c`, storing the result in array `a`, and repeats this operation for `*size` elements. If the loop bound `*size` does not change during execution, LICM optimizes the code by hoisting the dereference of `*size` outside the loop, as shown in Figure 2.2b. It introduces a temporary variable `t` to hold the loop bound and rewrites the loop to use `t` instead. Thus, avoiding repeated

loading of the loop bound on every iteration and enhancing efficiency.

Dead Store Elimination (DSE). DSE [1] targets store operations that assign values to memory locations that are never subsequently read. Such dead stores consume processor time and memory bandwidth unnecessarily. This optimization identifies and eliminates them to reduce resource usage and improve overall performance.

```
void foo(int *a) {
1.  *a = 10;
2.  *a = 20;
}
```

(a) The original code.

```
void foo(int *a) {
3.  *a = 20;
}
```

(b) The transformed code.

Figure 2.3: An example demonstrating how DSE can optimize the code.

In the example shown in Figure 2.3a, line 1 writes a value to the memory location pointed to by `a`, but that value is immediately overwritten by line 2 without being read in between. Since the first store has no observable effect on the program's behavior, a compiler with DSE will remove line 1 during optimization, resulting in the code in Figure 2.3b. DSE reduced an unnecessary store/write operation that might result in performance improvement.

Loop Vectorization (LV). Loop vectorization [4] transforms scalar loop operations into vector operations that can execute in parallel using SIMD (Single Instruction, Multiple Data) instructions. The compiler rewrites the loop to operate on multiple data elements per iteration, significantly reducing execution time, especially for compute-intensive workloads.

The example shown in Figure 2.4a performs element-wise addition of arrays `b` and `c`, storing the result in array `a`, one element at a time within a loop. When the compiler can determine that the pointers `a`, `b`, and `c` refer to disjoint memory regions (i.e., they do not alias), it safely assumes that no write to `a[i]` interferes with reads from `b[i]` or `c[i]`. This enables loop vectorization. In the transformed code (Figure 2.4b), the main loop processes four

```
void foo(int *a, int *b,
    int *c, int *size) {
1. for(int i = 0;
     i < *size; i++)
2.  a[i] = b[i] + c[i];
}
```

(a) The original code.

```
void foo(int *a, int *b,
    int *c, int *size) {
3.  int t = *size;
4.  int i;
5.  for (i = 0; i+3 < t;
      i = i+4)
6.    a[i:i+3]
       = b[i:i+3] + c[i:i+3];
7.
8.  for (; i < t; i++)
9.    a[i] = b[i] + c[i];
}
```

(b) The transformed code.

Figure 2.4: An example to demonstrate how LV can optimize the code.

elements at a time, significantly improving performance. A cleanup loop then handles any remaining iterations.

Superword-Level Parallelism Vectorizer (SLP). SLP [33, 34] vectorization identifies independent, isomorphic operations within a basic block. These operations are grouped together and converted into vector instructions, allowing them to be executed simultaneously. This fine-grained form of vectorization complements loop vectorization by exploiting parallelism even in non-loop code.

```
void foo(int *a, int *b) {
1.  b[0] = a[0] * 2;
2.  b[1] = a[1] * 2;
}
```

(a) The original code.

```
void foo(int *a, int *b) {
3.  b[0:1] = a[0:1] * 2;
}
```

(b) The transformed code.

Figure 2.5: An example to demonstrate how SLP can optimize the code.

The code in Figure 2.5a multiplies two consecutive elements of array `a` by 2 and stores the results in array `b`. If `a` and `b` do not overlap, the compiler can safely apply SLP vectorization. SLP groups independent, adjacent scalar operations into a single vector instruction that operates on both elements simultaneously. This transformation reduces instruction count and improves data-level parallelism. As a result, lines 1 and 2 are merged into line 3 in Figure 2.5.

Chapter 3

Related Work

Alias analysis is a foundational research area in compiler optimization and program analysis, focused on determining whether two expressions in a program may refer to the same memory location. It plays a critical role in enabling optimizations such as dead store elimination, loop-invariant code motion, and vectorization. A central challenge in this area is balancing precision and scalability, especially for interprocedural analysis, where aliasing information must be propagated across function boundaries.

3.1 Classical and Foundational Static Techniques

One of the earliest interprocedural approaches [35] demonstrated how alias information could be propagated across procedures using a binding graph to represent formal parameter interactions, although it focused primarily on call-by-reference parameters. Subsequently, Andersen's inclusion-based analysis [28] and Steensgaard's unification-based approach [5] became two of the most foundational techniques in this space. Andersen's method is a flow-insensitive and context-insensitive analysis. It performs weak updates, which accumulate points-to information rather than overwriting it, making it less precise than flow- or context-sensitive analyses but still more precise than Steensgaard's unification-based method. Andersen's method is cubic in time, whereas Steensgaard's is fast (linear time). These trade-offs highlight the underlying contest between scalability and precision in alias analysis.

3.2 Enhancements for Precision and Scalability in Static Analyses

Several interprocedural analyses aim to improve upon the limitations of Steensgaard's and Andersen's techniques. Shapiro and Horwitz [36] enhanced Steensgaard's unification model by introducing a parameterized framework that replaces each node's single outgoing edge with multiple out-degrees, enabling more precise aliasing relationships through repeated fixpoint computations. In contrast, Andersen-style inclusion-based analysis has been extended [15, 37–40] by focusing on identifying and collapsing strongly connected components within the constraint graph to improve analysis efficiency without sacrificing inclusion semantics. Additional strategies to mitigate imprecision include the use of weak updates, which introduce a program call graph–based formulation [41], and on-demand alias reasoning [42–44]. A hybrid unification-based approach [45] was introduced that uses one-level flow sensitivity via directional assignments, improving precision at the top level of pointer chains while retaining the scalability of Steensgaard's analysis. It effectively achieves near-Andersen precision for top-level pointers by encoding directionality in pointer relationships. Type-based analyses have also been explored as a means to improve precision [46–48]. Despite the precision improvements, the gains from these approaches were limited due to the absence of context and flow awareness. Additionally, even scalability remained a concern.

Flow-sensitivity and context-sensitivity are two independent concepts that have been extensively employed to further improve the precision of alias analysis. Flow-sensitive analyses respect the control flow of a program and track how alias relationships evolve through program execution. Various flow-sensitive [49] analyses use different approaches to maintain scalability in addition to precision improvement. Choi and Burke used sparse evaluation graphs to model how alias sets change across control paths [9], and Hind and Burke tracked entry and exit sets at the statement level across control-flow boundaries [6]. These ideas were

advanced by introducing partial transfer functions [12], symbolic pointer analysis with BDDs [13], and semi-sparse analysis with BDDs [50] to summarize alias effects in a flow-aware manner.

Later, def-use chain-based approaches [51–53] explicitly captured the flow of data (or pointer values) between variables. Other approaches [14, 54–57] use an auxiliary analysis to compute def-use chains and then create efficient representations of the chains. Several recent approaches [42,58,59] compute def-use chains on demand, significantly reducing unnecessary propagation. SkipFlow [16] further advances this line of work by introducing a predicated points-to analysis that interprocedurally tracks the flow of both primitives and objects using a data structure called a predicated value propagation graph (PVPG), explicitly modeling control-flow branching using predicate edges. Despite these innovations, flow-sensitive analyses are still underutilized in production compilers due to their higher memory demands and implementation complexity, making them challenging to deploy at scale for large codebases.

Several interprocedural and context-sensitive pointer analysis algorithms have been proposed. Examples include the call-string approach [8, 10, 13, 19], cloning-based strategies [7,11,52], and modular summary-based methods [9,12,18,20,60]. Recent approaches such as SVF [14] and FSAM [61] (for multi-threaded programs) achieve context-sensitivity using sparse value flow graphs and summarization techniques. These analyses offer superior precision but suffer from scalability limitations due to path explosion and memory requirements, particularly in recursive or large modular programs.

Several empirical studies [62–65] confirm that context-sensitive analyses outperform context-insensitive ones in terms of precision, especially when field sensitivity and flow sensitivity are also used. Hind et al. [66] provides a taxonomy of pointer analysis techniques and articulates the trade-offs clearly.

While numerous enhancements have improved static alias analysis in theory, it is also important to assess whether these precision gains even lead to real-world optimization benefits.

3.3 Assessing the Practical Impact of Improved Precision

Beyond scalability and precision, it is equally important to assess whether increasing alias analysis precision leads to meaningful improvements in optimization opportunities and tangible performance gains. While much of the literature focuses on algorithmic design and comparison, another line of work evaluates the practical impact of alias precision on compiler optimizations. One study computes a loose upper bound on potential benefits by assuming no pointer aliases, thereby estimating the theoretical maximum gain achievable with perfect alias information [21]. This highlights how conservative aliasing assumptions in real-world compilers can inhibit scalar optimizations. However, incorrect assumptions about no-alias relationships between pointer pairs can optimize the program in such a way that it leads to crashes at runtime, or incorrect or undefined behavior of the program.

More recently, Phulia et al. [22] exploits C's unspecified order of evaluation to statically infer must-not-alias relationships typically missed by traditional analyses. These additional disambiguations enable compilers to perform more aggressive optimizations by proving that certain memory accesses cannot interfere. This is a user-annotated approach. Thus, this approach is prone to human error and may focus on code regions that have minimal impact on performance.

ORAQL [67] is another optimistic approach that tries to estimate the gap between the current and ideal performance of HPC applications. The approach proposed in ORAQL [67] begins optimistically, treating every may-alias pointer pair as no-alias, then compiles and executes the program, and compares the

output with the expected program output. If the output matches, the optimistic assumption holds. If not (or the program results in an error), ORAQL iteratively tries to make the assumption less optimistic, using a bisection strategy, iteratively dividing the set of alias queries into two halves until all queries can be answered conclusively. ORAQL focuses primarily on HPC applications.

Weber et al. [68] argue against improving the precision of alias analysis by suggesting that only about 3% of alias information leads to beneficial optimizations. While there is merit to the idea, as indicated by some of our experiments (Section 4.4), our results indicate an overall performance benefit for most real-world benchmarks.

3.4 Towards Dynamic Analysis: Profiling-based Strategies and Runtime Pointer Disambiguation

Given the limited adoption of precise static techniques in production compilers, dynamic methods offer an alternative path to precision with a manageable overhead. Our work takes a dynamic analysis-based approach to identify actual alias relationships that occur at runtime. By instrumenting the code, we capture fine-grained alias behavior during execution, enabling us to quantify not only the realizable performance improvement from disambiguation but also to pinpoint the specific alias relationships that most impact optimization decisions. This methodology is inspired by prior work that employed instrumentation and dynamic analysis to detect bugs in alias analysis implementations [69, 70]. Although such techniques do not scale well to large, real-world applications due to the overhead of instrumentation, they are highly effective for studying which aliasing behaviors most influence optimization potential, guiding future analysis design toward the most performance-critical cases. As with any profiling-based approach, the results reflect only the behaviour observed at runtime, and we rely on users to profile with representative inputs so that the conclusions are meaningful for their workloads.

While these alias analyses estimate the limits of performance gains, another class of approaches provides a more grounded perspective by disambiguating pointers at runtime. These methods aim to estimate actual performance benefits under realistic conditions and typically rely on lightweight runtime techniques [17, 23, 24, 26, 27], combining code versioning with statically generated dynamic checks. Campos et al. [23], for instance, implements code versioning at the function granularity, generating multiple versions of a function where the optimized variant assumes non-overlapping pointer arguments. At each call site, a combination of static analysis and dynamic checks determines the version to execute based on the aliasing relationships among function parameters.

Expanding upon function-level code versioning, other works extend this idea to loop-level transformations, which is particularly impactful since loops typically dominate execution time. These techniques insert dynamic checks on memory accesses within loops to determine, at runtime, whether aggressive optimizations can be safely applied. Alves et al. [24] utilizes polyhedral analysis [71–73] and symbolic range analysis [74–76] to generate $O(1)$ dynamic checks that verify the non-overlapping nature of memory accesses. Similarly, the LLVM compiler leverages Scalar Evolution analysis [77, 78] to support dynamic alias checks in loop versioning. These techniques are particularly effective when loop bounds are loop-invariant, enabling efficient static reasoning.

To handle more general cases—specifically loops where bounds are not loop-invariant—Alves et al. [24] proposed a purely dynamic approach. This technique disambiguates pointers at runtime by identifying the base object that a pointer refers to. If two pointers map to distinct objects, they are guaranteed not to alias. The method maintains a red-black tree of memory ranges for allocated objects and performs $O(\log n)$ lookups to determine object identities. While general and precise, this method introduces significant overhead in practice due to the cost of maintaining and querying dynamic metadata structures. Our approaches, discussed in Chapters 5 and 6, address this limitation by re-

ducing the overhead of dynamic checks to constant time ($O(1)$), making them more suitable for real-world use.

Complementing these techniques, our approach in Chapter 5 uses a profiler to identify loops that benefit from optimizations, while Chapter 6 leverages profiling to determine which dynamic checks are necessary. Both strategies aim to minimize overhead by focusing runtime checks only on performance-critical and alias-sensitive regions. The idea of profile-guided speculative optimization has been explored in prior work [79–84], where optimized code is generated under non-alias assumptions and guarded by dynamic checks. When assumptions fail, execution falls back to recovery code to maintain correctness. These techniques were highly effective on older hardware with high memory latency, but their benefits diminish on modern processors with larger caches and sophisticated prefetching. As such, they are most effective when recovery is rare and profiling indicates high confidence in alias assumptions.

To further reduce dynamic check overhead, our approach also draws on interference-based strategies, similar to those used in register allocation. We construct an interference graph where nodes represent memory accesses and edges represent potential conflicts. Just as registers are assigned using graph coloring, we assign regions in our analysis based on coloring the interference graph [85]. Although graph coloring is NP-complete, we adopt a widely used approximate coloring algorithm [86] to ensure efficiency. While linear scan register allocation [87–90] is a fast alternative, it depends on program control flow, which our analysis does not leverage—making it unsuitable in our context.

Chapter 4

Estimating the Impact of the Improved Alias Analysis Precision on Program Performance

Static alias analysis is typically evaluated along two key dimensions: scalability and precision. Due to the high computational cost of precise analyses, production compilers often prioritize scalability, adopting conservative approximations that avoid aliasing ambiguities. While this ensures robustness on large codebases, it limits the compiler's ability to apply aggressive optimizations that rely on accurate alias information. Although improving precision can potentially unlock such optimizations, it is equally important to assess whether these improvements result in meaningful performance gains in practice.

Chowdhury et al. [21] estimate the potential benefits of perfect alias information by assuming that no pointer pairs alias. While this approach provides insight into the estimate of the performance upper bound, it does so without regard to correctness, and therefore may introduce unsafe transformations that would crash or miscompile real programs. More importantly, it fails to provide a realistic understanding of how alias imprecision impacts the performance of real-world applications. ORAQL [67] is another optimistic approach that tries to estimate the gap between the current and ideal performance of HPC applications. Here, ideal performance is said to be achieved in the presence of highly precise no-

alias information.

We propose a profiling-based tool that estimates the practical performance benefits of improved alias analysis. Our approach dynamically identifies alias relationships missed by static analysis and annotates the program with this information. These annotations increase precision and allow the compiler to apply optimizations that would otherwise be blocked. This enables an empirical evaluation of how alias precision affects performance in both sequential and multi-threaded programs. Additionally, developers can use the inferred annotations as feedback to safely insert the `restrict` keyword in cases where non-aliasing is guaranteed for given inputs but not inferred statically. These annotations are not applied automatically. Instead, they help knowledgeable users reason about aliasing behavior and make informed, correctness-preserving decisions. We also expose cases of inefficient code generation caused by overly aggressive optimizations (Figure 4.6), helping compiler developers identify and fix optimization bugs.

While annotations can be provided manually or generated automatically, user-driven approaches such as the approach proposed by Phulia et al. [22] are prone to human error and may focus on code regions that have minimal impact on performance. To overcome these challenges, our tool automatically infers alias annotations by instrumenting and executing the program. The inferred annotations are presented to the user as feedback, allowing them to validate or discard those that are input-specific or less reliable. In practice, the feedback can be automatically filtered to performance-critical regions, avoiding exhaustive manual inspection and keeping the approach practical for large applications. These annotations can be used to annotate an unoptimized program or debug existing user-provided annotations.

Beyond annotating alias information, our tool identifies the functions and the optimizations that contribute most to performance improvements. This capability

is a direct consequence of estimating alias precision bounds and is not provided by existing compiler analyses. This helps developers make informed decisions about where to apply more precise static analyses or validate annotations. We implement our approach in an LLVM-integrated tool named Horizon+.

Although ORAQL and Horizon+ share the same overall goal, they differ in several important ways. ORAQL uses a bisection strategy to separate pointer pairs that do not alias from the ones that may alias, and is an optimistic or assumption-driven method, whereas Horizon+ uses a profiling-based approach to infer must and no-alias relationships at runtime and thus identifies true alias relationships, which hold true only for the profiled inputs. Thus, unlike ORAQL, Horizon+ can also handle must-alias queries. ORAQL relies on repeated compile-and-run iterations ($2n-1$ iterations in the worst case), whereas Horizon+ requires just two compile-and-run iterations, and the runtime depends only on the number of queries, irrespective of their result and order. Furthermore, ORAQL focuses primarily on HPC applications (tested on a maximum of 50 KLOC), whereas Horizon+ focuses on CPU-intensive workloads (tested on a maximum of 1304 KLOC).

This chapter essentially provides answers to the following questions:

RQ1 For a program and the given input test cases, what is the estimated upper bound on the precision improvement potential of alias analysis?

RQ2 How does the estimated range of precision improvement impact the performance of the program for given input test cases?

RQ3 Can Horizon+ precisely identify functions for performance improvement when compiled using input-specific alias annotations?

RQ4 Which optimizations benefit the most from precise input-specific alias in-

formation for the given program?

This chapter makes the following contributions:

- We present a method to automatically infer and annotate precise alias relationships within a program.
- We integrate the technique into LLVM and evaluate it on standard benchmarks to estimate the improvement potential of the default alias analysis. We also identify the optimizations that benefit most from improved precision. The implementation is packaged into a new LLVM-integrated tool named Horizon+.
- We measure the performance improvements on Polybench, CPU SPEC 2017, and SPLASH-2 benchmarks when enhanced alias information is introduced into the compilation process.

Note that Horizon+ does not use runtime checks. It annotates the program only for the alias relationships that certainly hold while profiling. For pointers that overlap even slightly, as mentioned in the Section 1.2, it uses conservative `may-alias` estimates. As a result, Horizon+ does not apply optimizations that are valid for only a majority of executions. Such cases are intentionally excluded to avoid unsound transformations.

4.1 Motivating Example

We now present an overview of our approach using the example in Figure 4.1a, which is taken from the `kernel_bicg` function in the `bicg` benchmark from the Polybench suite. This function implements the biconjugate gradient stabilized method for solving nonsymmetric linear systems. The snippet contains a nested loop that updates arrays s and q in each iteration. For the default input test case, the arrays s, q, r, p, and A are passed as arguments and do not overlap in memory.

```
1. static void kernel_bicg(
    int m, int n,
    double A[][], double s[],
    double q[], double p[],
    double r[]) {
2.  for (int i = 0;
     i < _PB_N; i++) {
3.   q[i] = SCALAR_VAL(0.0);
4.   for (int j = 0;
      j < _PB_M; j++) {
5.    s[j] = s[j]
6.     + r[i] * A[i][j];
7.    q[i] = q[i]
8.     + A[i][j] * p[j];
     }
    }
}
```

(a) Original code.

```
9. static void kernel_bicg
     (...) {
10.  for (int i = 0;
      i < _PB_N; i++) {
11.   q[i] = SCALAR_VAL(0.0);
12.   double tmp = q[i];
13.   for (int j = 0;
       j < _PB_M; j++) {
14.    s[j] = s[j]
        + r[i] * A[i][j];
15.    tmp = tmp
        + A[i][j] * p[j];
      }
16.   q[i] = tmp;
     }
}
```

(b) Optimized code.

```
17. static void kernel_bicg
     (...) {
18.  for (int i = 0;
      i < _PB_N; i++) {
19.   q[i] = SCALAR_VAL(0.0);
20.   for (int j = 0;
       j < _PB_M; j++) {
21.    s[j] = s[j]
        + r[i] * A[i][j];
22.    q[i] = q[i]
        + A[i][j] * p[j];
23.    Instrument(&s[j], &r[i]);
24.    Instrument(
        &s[j], &A[i][j]);
25.    Instrument(&s[j], &q[i]);
26.    Instrument(&s[j], &p[j]);
27.    Instrument(&q[i], &r[i]);
28.    Instrument(
        &q[i], &A[i][j]);
29.    Instrument(&q[i], &s[j]);
30.    Instrument(&q[i], &p[j]);
     }
    }
}
```

(c) Instrumented code after annotation inference phase.

```
31. static void kernel_bicg
     (...) {
32.  for (int i = 0;
      i < _PB_N; i++) {
33.   q[i] = SCALAR_VAL(0.0);
34.   for (int j = 0;
       j < _PB_M; j++) {
35.    s[j] = s[j]
        + r[i] * A[i][j];
36.    q[i] = q[i]
        + A[i][j] * p[j];
37.    noalias(&s[j], &r[i]);
38.    noalias(
        &s[j], &A[i][j]);
39.    noalias(&s[j], &q[i]);
40.    noalias(&s[j], &p[j]);
41.    noalias(&q[i], &r[i]);
42.    noalias(
        &q[i], &A[i][j]);
43.    noalias(&q[i], &s[j]);
44.    noalias(&q[i], &p[j]);
     }
    }
}
```

(d) Annotated code after annotation incorporation phase.

Figure 4.1: An example to discuss Horizon+ approach.

The absence of aliasing between s and q enables optimizations such as Global Value Numbering (GVN) and Loop Invariant Code Motion (LICM). As shown in Figure 4.1b, GVN can hoist the memory access to $q[i]$ out of the inner loop, and LICM can introduce a temporary variable tmp to accumulate intermediate values and move the store to $s[j]$ outside the loop. These optimizations reduce two memory accesses from the loop body, improving performance.

However, the default alias analysis in LLVM is intra-procedural and conservative. It lacks the context to determine whether s and q alias and therefore assumes a `may-alias` relationship. As a result, the compiler cannot safely apply these optimizations due to the absence of deterministic aliasing information for function arguments.

Our approach addresses this limitation by profiling the application to infer alias relationships that cannot be determined statically. These inferences are specific to the input test cases provided. As shown in Figure 4.1c, during the first phase, instrumentation is inserted for pointer pairs such as ($s[j]$, $r[i]$), ($s[j]$, $A[i][j]$), ($s[j]$, $q[i]$), ($s[j]$, $p[j]$), ($q[i]$, $r[i]$), ($q[i]$, $A[i][j]$), ($q[i]$, $s[j]$) and ($q[i]$, $p[j]$). Each call to `Instrument(`x`,` y`)` represents dynamic tracking of the alias relationship between the two memory locations. The program is then executed, and the resulting alias information is stored in a log file.

In the second phase, Horizon+ uses this log file to annotate the program with the inferred alias relationships. Figure 4.1d shows the annotated version of the code. The annotation `noalias(&`$s[j]$`, &`$q[i]$`)` explicitly informs the compiler that these two memory accesses do not alias. This added information enables the compiler to apply optimizations that were previously missed due to lack of precision. It is important to note that these inferred annotations are valid only for the input test cases used during profiling.

In this context, our goal is not to provide universally sound annotations for

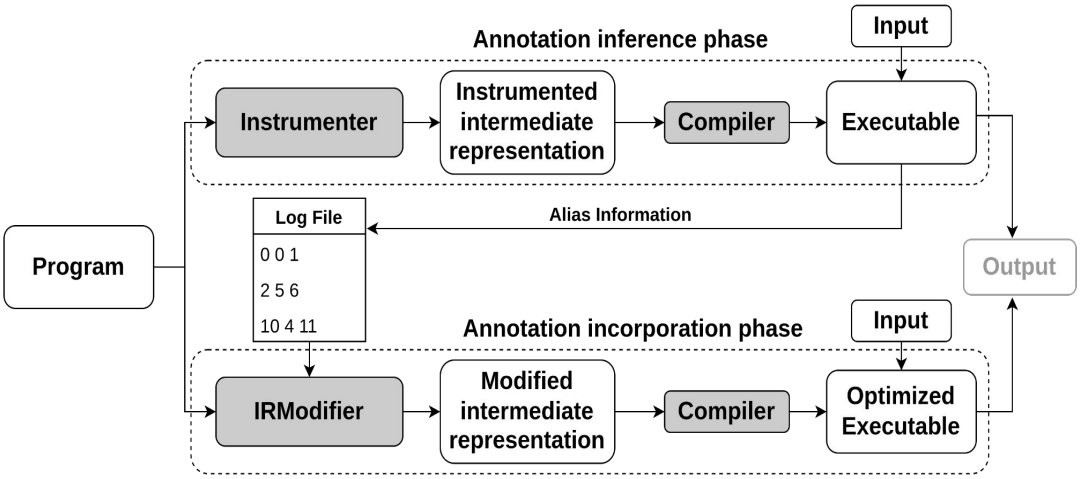


Figure 4.2: Architecture of Horizon+.

all possible executions, but rather to estimate the upper bound on the precision improvements achievable by alias analysis and to understand their impact on performance. Consequently, Horizon+ is not designed as a general-purpose optimization framework, instead, it serves as a tool for developers and compiler designers to identify missed optimization opportunities and reason about the limitations of current alias analysis techniques.

4.2 Horizon+

Figure 4.2 shows the architecture of Horizon+. It executes the program to infer alias information missed by static alias analysis and annotates the program with the newly discovered information. These annotations enable the compiler to apply additional optimizations that were previously blocked due to conservative aliasing assumptions. Horizon+ operates in the following five phases:

- **Annotation inference phase:** This phase takes the optimized intermediate representation (IR) of the program as input. Horizon+ instruments the IR to track alias relationships missed by the static analyzer, compiles it into

an executable, and runs it with default input test cases. During execution, a log file is generated, recording the dynamically inferred alias relationships.

- **Annotation incorporation phase:** The optimized IR and the log file from the previous phase are used to annotate the IR with the inferred alias information. The annotated IR is then recompiled. The compiler leverages these annotations during optimization to generate more efficient code.
- **Precision Analysis:** In this phase, Horizon+ compares the native and annotated IRs by compiling both and evaluating the precision of the alias analysis. Section 4.4.2 presents the estimated precision improvement potential of LLVM's default alias analysis on standard benchmarks.
- **Performance Analysis:** This phase compares the performance of the native and optimized executables generated in the annotation incorporation phase. Horizon+ runs both versions on default input test cases and reports the performance improvements and helps in identifying the optimizations that benefit the most, as discussed in Section 4.4.3.
- **Feedback to the user:** In the final phase, benchmarks that show significant performance improvements are further analyzed to identify the most impacted functions. Using the Gprof tool [91], Horizon+ maps the relevant IR annotations back to source-level functions. These annotations are reported to the user for validation, allowing them to retain those likely to hold across different inputs.

4.2.1 Annotation inference phase

The goal of the annotation inference phase is to dynamically infer precise alias annotations for pointer pairs that static alias analysis fails to disambiguate. To achieve this, Horizon+ maintains a list called the *tracking-list*, which includes all pointers that may require runtime tracking.

Initially, Horizon+ adds the pointer operands of loads, stores, atomic instructions, and memory intrinsics (e.g., `memcpy`) to the *tracking-list*. These operands

```
struct MetaDataTy {
  unsigned long long offset:48,
  unsigned long long alias:1
};
Initialize() {
  for each pointer pair (P1, P2) {
      MetaData[P1, P2].offset = MAX_OFFSET;
      MetaData[P1, P2].alias = 1;
  }
}

Instrument(P1, P2) {
    offset = abs(P1 - P2);
    MetaData[P1, P2].alias &= (offset == 0);
    if (offset < MetaData[P1, P2].offset)
        MetaData[P1, P2].offset = offset;
}
```

Figure 4.3: Annotation inference logic. The minimum absolute relative offset between P1 and P2 is stored at runtime. The alias field in the metadata is set to zero if there is at least one instance when P1 and P2 are not equal.

represent either addresses of array elements or internal fields when the array contains structures. It then computes the base address of each element by recursively analyzing pointer arithmetic and typecasts, and adds these base addresses to the list. Additionally, pointer-type function arguments are also included. Pointer pairs where neither pointer dominates the other in control flow are excluded, as such cases are rare in optimization queries and have minimal impact.

Instrumentation logic. Horizon+ inserts instrumentation for each unordered pointer pair P1, P2 in the *tracking-list* if static alias information is unavailable for that pair. For each function, it allocates a `MetaData` array of size $N(N-1)/2$, where N is the number of tracked pointers. Each pointer pair is assigned a unique index in this array.

Figure 4.3 shows the annotation inference logic. For each instrumented pointer pair {P1, P2}, Horizon+ tracks the minimum absolute relative difference between P1 and P2. In addition, Horizon+ tracks whether a pointer pair always aliases by observing their runtime behavior during profiling. A pair is classi-

fied as `must-alias` only if the two pointers are observed to overlap at every access throughout execution, providing a guarantee with respect to the profiled inputs. This is required for the case when P1 and P2 both alias and do not alias (at different times) during the program execution (`may-alias`). At the entry of the `main` routine, the `Initialize` routine is called. It sets the `offset` and `alias` fields in the metadata to the maximum possible offset (`MAX_OFFSET`) and one, respectively, for each unordered pointer pair in the *tracking-list*. Before exiting the program, Horizon+ logs the metadata to a file that is used by the annotation incorporation phase (Section 4.2.2).
The log file entry corresponding to pointer pair {P1, P2} is interpreted as follows:

1. If `offset = 0` and `alias = 1`, then P1 and P2 are aliases.
2. If (`offset = 0` and `alias = 0`) or `offset = MAX_OFFSET`, then P1 and P2 may or may not alias.
3. If `offset > 0`, then P1 and P2 either partially alias or do not alias.

4.2.2 Annotation incorporation phase

During the annotation incorporation phase, Horizon+ annotates the IR with the alias information present in the log file generated during the inference phase (Section 4.2.1). If an unordered pair of pointer expressions {P1, P2} is inferred as must-alias, and the instruction computing P1 dominates the instruction computing P2, Horizon+ replaces all uses of P2 with P1. If the types of P1 and P2 are different, P2 is replaced after typecasting P1 to P2's type. After this step, the compiler can statically infer that all usages of P2 and P1 are pointing to the same address. However, the `no-alias` cases are tricky. We rely on the metadata and intrinsics for inferred `no-alias` pointer pairs that are used in any memory access instruction, described in Section 4.3. Consider the `checkAlias` function in Figure 4.4. `checkAlias` is the original alias analyzer interface for alias queries.

```
checkAlias(void *P1, void *P2, size_t P1_size,
    size_t P2_size);

newCheckAlias(void *P1, void *P2,
    size_t P1_size, size_t P2_size) {
    Ret = checkAlias(P1, P2,
        P1_size, P2_size);
    if (Ret == DONT_KNOW)
        for each metadata intrinsic
            (X, Y, offset)
            if ((P1 == X || P2 == X)
            && (P1 == Y || P2 == Y))
                if (offset >=
                max(P1_size, P2_size))
                    return NOT_ALIAS;
                else
                    return PARTIAL_ALIAS;
    return Ret;
}
```

Figure 4.4: The original `checkAlias` interface takes a pointer pair and their respective sizes as inputs. The `newCheckAlias` routine, if required, also walks all the metadata intrinsics to check if an intrinsic contains the input pointer pair and an offset that is greater than the sizes of both the pointers.

The `checkAlias` function takes two pointers `P1` and `P2`, and their sizes `P1_size` and `P2_size` as input, where `P1_size` and `P2_size` should be statically determined. The `checkAlias` function reports a pointer pair to be `no-alias`, if it can statically infer that the memory regions ``P1, P1 + size_P1'' and ``P2, P2 + size_P2'' do not overlap. To answer these queries, Horizon+ annotates the source code with metadata intrinsics. A metadata intrinsics is ignored by the compiler during the code generation but retained during the intermediate program transformations. Our metadata intrinsic contains a pair of pointers and the minimum absolute relative offset observed during the inference phase (Section 4.2.1).

We have modified the `checkAlias` interface to `newCheckAlias` (see Figure 4.4). `newCheckAlias` additionally iterates the metadata intrinsics to find an answer if the `checkAlias` does not have a definite answer. The prototype of `newCheckAlias` is the same as `checkAlias`. If `newCheckAlias` finds metadata intrinsic that contains P1 and P2 and the offset is greater than

or equal to the maximum of `P1_size` and `P2_size`, it reports them as `no-alias`. Note that the offset is the minimum absolute relative offset observed during the annotation inference phase. If `offset >= max(P1_size, P2_size)`, then the memory regions ``P1, P1+P1_size'' and ``P2, P2+P2_size'' do not overlap. However, this is not the best answer we can give. For example, consider a case when `P1_size` is less than `P2_size`. At runtime, the minimum relative offsets are `P1_size` if `P1 < P2` and `P2_size` if `P2 < P1`. In this case, if the offset is greater than equal to `P1_size`, we can safely conclude that P1 and P2 are `no-alias`. To support this, we need to track two offsets during the annotation inference phase. The first one is the minimum relative offset if `P1 < P2`, and the second one is the minimum relative offset if `P2 < P1`. We have not implemented this feature. To estimate the loose upper bound of the precision loss due to this restriction, we counted the number of queries in which the runtime minimum offset lies between the size of input pointer pairs. The maximum number of such queries is 88 for the `500.perlbench_r` benchmark. Notice that even if we log two offsets during the annotation inference phase, some of these queries may remain unanswered.

4.3 Implementation

We implemented Horizon+ as a part of the LLVM infrastructure (v10.0.0). The default implementation of alias analysis in LLVM involves four passes: `basic`, `type-based`, `scoped no-alias`, `globals mod/ref` and, `scalar evolution` alias analysis. For an alias query, these analyses are performed in a sequence until a definite answer is reached. In LLVM, the alias analysis results are used for optimizations such as GVN, LICM, DSE, LV, and SLP.

Horizon+ works on an optimized intermediate representation (IR) of the program. The input to the annotation inference phase (Section 4.2.1) and annotation incorporation phase (Section 4.2.2) is an optimized LLVM IR compiled with the `O3` optimization level. We disable SLP, LV, and loop unrolling while generating the input IR. Although the annotation incorporation pass operates on

the original program structure, some loops may already be vectorized or unrolled by earlier optimization passes. In such cases, additional alias information generated using the annotation inference phase does not enable further optimization by vectorization or loop unrolling. However, if loop unrolling and vectorization are first performed on an IR after the additional alias information is incorporated, LLVM can vectorize and unroll better than it does without the additional alias information. So, it is better to disable these optimizations during the initial optimization and only enable them later, when annotations have been incorporated.

We have added two new passes to LLVM. The first pass, known as "dynaa", performs the annotation inference phase. The second pass, known as "get-dynaa", performs the annotation incorporation phase. Horizon+ also annotates the source code with alias information, in addition to providing statically identified aliases of pointer pairs as feedback to the user. To get the inferred annotations, the "map-annotations" option can be provided with the annotation incorporation pass. This feedback allows users to incorporate the chosen annotations at the source code level. They can then compile and execute the source code on different compilers. The annotations provided as feedback to the user are mapped to the source code by LLVM using the debug information added by the option (`-g`). This information is not guaranteed to be correct for some cases, as mentioned in [92]. The debug information might get affected due to the transformations and optimizations taking place due to different LLVM passes. However, in most cases, the information will be accurate.

The annotation inference pass selectively instruments the functions that contribute significantly (>95%) to the total execution time of the benchmark. The per-function execution time is obtained using Gprof [91] tool. The annotation incorporation phase incorporates the information (generated during the annotation inference) from the log file phase into the IR. The incorporation of `must-alias` information is simple and has been described in Section 4.2.2.

The remaining information (`no-alias`) present in the log file is added using an LLVM metadata and intrinsic. The resulting IR is further provided to the other optimization passes to leverage the information collected by Horizon+. Whenever the default implementation of alias analysis fails to provide a definitive answer for a pointer pair, the result is checked in the newly added information by parsing through the metadata intrinsics (`newAliasCheck` routine in Figure 4.4). In this case, the result can be either `no-alias` or `may-alias`.

Adding `no-alias` information. We rely on the `scoped no-alias analysis` pass of LLVM by attaching the `alias.scope` and `noalias` metadata [93] to inferred `no-alias` pointer pairs that are used in any memory access instruction. The `alias.scope` and `noalias` metadata is attached to such non-overlapping memory accesses. Currently, the `scoped no-alias analysis` pass of LLVM provides metadata support for the limited set of instructions (i.e. memory access instructions). Thus, to incorporate the information about `no-alias` pointer pairs other than the memory access instructions, we also modify the LLVM's backend to add a new LLVM intrinsic.

LLVM intrinsics can preserve metadata for the program throughout the transformation passes. We have added a new intrinsic named "llvm.noalias.dynaa" to the LLVM backend. This intrinsic takes three arguments. The first two arguments are pointers wrapped in the form of "LLVM's metadata nodes," and the third argument is a 64-bit integer. This type of intrinsic can also be referred to as a debug intrinsic since the arguments passed to this intrinsic hold the metadata for the program. The program needs to be compiled in debug mode (-g option), since the arguments of the intrinsic represent debug information. A sample usage of the intrinsic is this:

```
call void @llvm.noalias.dynaa(metadata DATA_TYPE %1,
metadata DATA_TYPE %2, metadata i64 c)
```

where `metadata` represents the LLVM's metadata node, `DATA_TYPE` represents the type of the pointer, `i64` represents a 64-bit integer, %1 and %2

represent pointers, and `c` represents a constant integer value. We refer to this intrinsic using *noalias-intrinsic*.

4.4 Evaluation

We evaluated Horizon+ on the default implementation of alias analysis in LLVM using 30 Polybench [94] benchmarks, eight C benchmarks which are available as a part of Intrate and FPrate suites in CPU SPEC 2017 [95], and 10 SPLASH-2 [96] benchmarks. The maximum program size in Polybench is approximately 1086 LOC, in CPU SPEC 2017 is approximately 1304 KLOC, and in SPLASH-2 is approximately 11760 LOC, demonstrating that our evaluation includes both small kernels and large real-world applications. For Polybench benchmarks, the extra-large input set has been used for the evaluation. For the CPU SPEC 2017 benchmarks, we have used the reference input set. Table 4.1 describes the input set used for three benchmarks of SPLASH-2 benchmark suite and for the remaining seven benchmarks we used the native input set mentioned in [97]. In order to understand the impact on the performance, benchmarks need to have reasonably high execution times. Therefore, we have used custom inputs to ensure reasonably high execution times for three benchmarks mentioned in Table 4.1. For simplicity, we refer ocean (contiguous_partitions) as `ocean_cp`, ocean (non_contiguous_partitions) as `ocean_ncp`, lu (contiguous_blocks) as `lu_cb` and lu (non_contiguous_blocks) as `lu_ncb`. The experiments were performed on 2.10GHz Intel(R) Xeon(R) Silver 4116 CPU 12 core machine with an x86_64 architecture and 32 GB primary memory which uses a 64-bit Ubuntu 18.04.1 operating system. We had disabled the hyper-threading during our experiments. For performance numbers, we used the minimum runtime out of the five runs for each benchmark.

Since ORAQL and Horizon+ largely have the same goal, we tried to evaluate and compare the performance of both tools on the selected benchmarks. However, we were unable to gather results for ORAQL because the artifacts for the tool were only partially available, leading to segmentation faults for the selected

Table 4.1: Input set description for SPLASH-2 benchmarks.

Benchmark	Input set
ocean_ncp	4098 x 4098 Grid, Timestep = 28800, Distance between grid points = 20000, Error tolerance = 1e-07
radiosity	Largeroom, BF refinement = 0.00025
raytrace	Environment file = car.env, Antialiasing with subpixels = 32768

benchmarks (CPU SPEC 2017).

4.4.1 Overhead of annotation inference phase

The annotation inference phase infers the annotations by instrumenting the program as described in Section 4.2.1. The added instrumentation significantly affects the execution time of the program. The overhead depends on the number of pointer pairs instrumented. We observed that the overhead of the annotation inference phase has a positive correlation with the number of pointer pairs instrumented. The overhead lies in the range of 1x (seidel-2d) to 126x (lu_cb) with an average overhead of 25x (with respect to the original execution time). 31 of the 48 tested benchmarks have an overhead of less than 25x. Even though the overhead is high, this is a one-time overhead that is incurred before the program is optimized and used in production.

4.4.2 Precision improvement potential

RQ1 For a program and the given input test cases, what is the estimated upper bound on the precision improvement potential of alias analysis?

The graph shown in Figure 4.5 plots the percentage of precision improvement potential which is computed as the percentage increase in the number of `no-alias` and `must-alias` pointer pairs when the Polybench, CPU SPEC 2017 and SPLASH-2 benchmarks are compiled with and without the program annotations. We use the LLVM's "aa-eval" pass to generate this information. For Polybench, the graph depicts benchmarks for which the precision improvement potential is greater than 10%. Out of these nine benchmarks, three benchmarks report an improvement potential of more than 30%. These results indicate

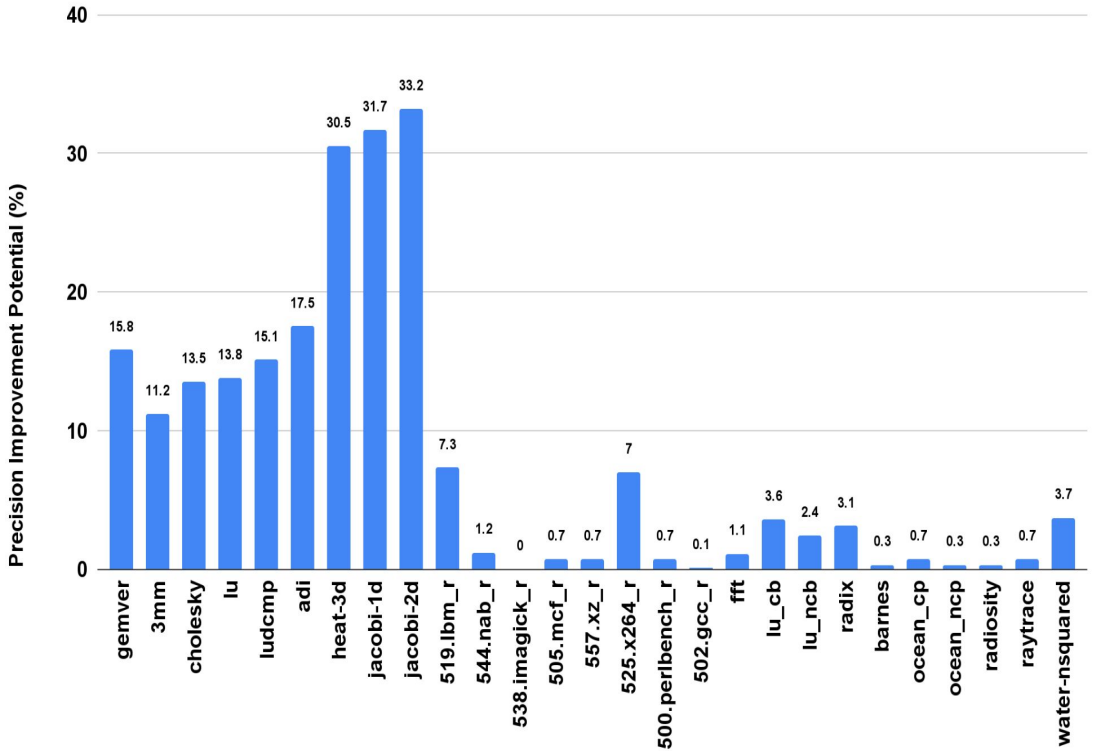


Figure 4.5: Precision improvement potential (%) for Polybench, CPU SPEC 2017 and SPLASH-2 benchmarks.

a substantial potential to improve the precision of the alias analysis implementation for the default input test cases.

We further observe that the current implementation of static alias analysis in LLVM misses pointer pairs involving:

- Inter-procedural information such as function arguments or return values of functions and,
- Information depending on user input (dynamic information) or conditional statements.

The information related to function arguments and return values of functions can be determined statically by performing inter-procedural analysis. However, due to its expensive nature, most alias analysis implementations avoid performing inter-procedural analysis. We observed that most pointer pairs missed

Table 4.2: Performance impact (%) comparison for Polybench and CPU SPEC 2017 benchmarks (AO = All optimizations enabled, MEO = The most effective combination of optimizations enabled).

Benchmark	AO (%)	MEO (%)
gemver	-1.02	0.2
gesummv	46.59	47.38
bicg	53.16	53.32
mvt	-0.37	0.36
adi	-7.67	0.33
jacobi-1d	20.47	21.11
519.lbm_r	1.15	1.15
544.nab_r	0.77	1.02
538.imagick_r	0.94	1.17
505.mcf_r	2.06	3.65
557.xz_r	1.64	2.1
525.x264_r	-1	11.56
500.perlbench_r	0.51	1.02
502.gcc_r	3.48	3.82

for benchmarks such as `jacobi-2d`, `heat-3d`, `jacobi-1d`, `adi`, `3mm`, `cholesky`, `lu`, `ludcmp` and `gemver` involve function arguments. These pointer pairs were missed due to the absence of inter-procedural alias information, as shown in Figure 4.5. Horizon+ was able to provide precise answers for such pointer pairs. Therefore, the precision improvement scope for these benchmarks is significantly higher than the other benchmarks. A large majority of benchmarks (39 of 48) demonstrate a precision improvement scope between 0% and 10%. The remaining nine benchmarks show a very high (greater than 10%) precision improvement potential. These estimates depend on the input test cases provided during the annotation inference phase.

4.4.3 Performance improvement potential

RQ2 How does the estimated range of precision improvement impact the performance of the program for given input test cases?

We estimated the impact on the performance of the Polybench and CPU SPEC 2017 benchmarks by incorporating the program annotations inferred by Horizon+ for the provided input test cases. We did this by computing the per-

centage change in the execution time of each benchmark compiled with and without annotations. Since it was discovered that certain optimizations could harm performance, we also tried calculating the percentage improvement in execution time for each benchmark while keeping only the most effective combination of optimizations enabled. To ensure correctness, we verified that the annotated programs produce the same outputs as the original unannotated versions for the given inputs, none of the benchmarks crashed, and all executions completed successfully. Table 4.2 shows these results for the CPU SPEC 2017 benchmarks and for the Polybench benchmarks that result in a significantly high performance impact (positive or negative).

```
static void kernel_adi(int tsteps, int n, double u[],
double v[], double p[], double q[]) {
  ...
  for (j=1; j<_PB_N-1; j++) {
    p[i][j] = -c / (a*p[i][j-1]+b);
    q[i][j] = (-d*u[j][i-1]+ (SCALAR_VAL(1.0)
      + SCALAR_VAL(2.0)*d)*u[j][i]
      - f*u[j][i+1]-a*q[i][j-1])/(a*p[i][j-1]+b);
  }
}
```

Figure 4.6: An example (`adi`) with negative performance impact of -8%.

By examining the intermediate representation and statistics in Table 4.2, we make the following interesting observations:

- In `bicg`, for the code snippet in Figure 4.1b, the `no-alias` relationship inferred by Horizon+ between arrays s, r, p, q, and A enables GVN and LICM. GVN moves the load instruction for $r[i]$ outside the inner loop since $r[i]$ is never modified inside the loop. LICM moves the store instruction for $q[i]$ to the end block of the inner loop by storing the intermediate results for the operation into a temporary variable. The final result is stored into $q[i]$ when the inner loop has finished executing. These optimizations further unroll the loop body by a factor of 2.
- In `gesummv`, for the code snippet in Figure 4.7, the `no-alias` relationship inferred by Horizon+ between arrays tmp, A, B, x and y enables

```
static void kernel_gesummv(
  int n, double alpha,
  double beta, double A[],
  double B[], double tmp[],
  double x[], double y[]) {
  ...
  for (i = 0; i < _PB_N;
    i++) {
    tmp[i] = SCALAR_VAL(0.0);
    y[i] = SCALAR_VAL(0.0);
    for (j = 0; j < _PB_N;
      j++) {
      tmp[i] = A[i][j]
        * x[j] + tmp[i];
      y[i] = B[i][j]
        * x[j] + y[i];
    }
    y[i] = alpha
      * tmp[i] + beta * y[i];
  }
}
```

```
static void kernel_jacobi_1d(
  int tsteps, int n, double A[],
  double B[]) {
  for (i = 1; i < _PB_N - 1; i++)
    B[i] = 0.33333 * (A[i-1]
      + A[i] + A[i + 1]);
  for (i = 1; i < _PB_N - 1; i++)
    A[i] = 0.33333 * (B[i-1]
      + B[i] + B[i + 1]);
}
```

Figure 4.7: Some examples (`gesummv` and `jacobi-1d`) with positive performance impact of 47% and 21% respectively.

GVN and LICM. LICM moves the store instructions for the array elements $tmp[i]$ and $y[i]$ out of the body of the inner loop by storing the intermediate results into temporary variables. The final result for $tmp[i]$ is stored right after the inner loop finishes execution. GVN removes the load instruction for $tmp[i]$ as the value for $tmp[i]$ can be obtained from the temporary variable used earlier to store its value. These optimizations further enable SLP vectorization for the operations being performed in the inner loop body.

- In `jacobi-1d`, for the code snippet in Figure 4.7, the `no-alias` relationship inferred by Horizon+ between $A[i-1]$, $A[i]$, $A[i+1]$ and $B[i]$ allows GVN to remove the load instructions for $A[i-1]$ and $A[i]$ as these values are respectively equivalent to $A[i]$ and $A[i+1]$ from the previous iteration. This optimization results in reducing the loads present in the body of the inner loop from 3 to 1. The optimization is also performed for the second inner loop which loads $B[i]$, $B[i-1]$, and $B[i+1]$. The op-

timization further benefits loop vectorization to avoid performing redundant loads ($A[i-1]$, $A[i]$, $B[i-1]$ and $B[i]$) in the loop body.

- In `adi`, for the code snippet in Figure 4.6, the no-alias relationship inferred by Horizon+ between $p[i][j]$ and $q[i][j]$ results in triggering SLP vectorization. SLP vectorization should only be performed when similar types of operations are performed independently. Otherwise, it can end up harming the performance of the program. In this case, SLP vectorization ends up adding extra vector instructions inside the loop body, increasing the execution time of the benchmark due to the new no-alias information. Table 4.2 shows that the performance of adi improves by 0.31% when SLP vectorization is disabled during compilation.
- In 525.x264_r, `pixel_hadamard_ac` function is responsible in degrading the performance of the benchmark. The `no-alias` information inferred by Horizon+ for the function `pixel_hadamard_ac` enables SLP vectorization due to the removal of one load instruction by GVN. This ends up adding extra instructions to the intermediate representation, increasing the overall cost of the function. Disallowing GVN while compiling the program with annotations improves the overall performance of the benchmark by 12%.

We further observed that some benchmarks reported difference in the performance impact when compiled with all optimizations enabled as opposed to when compiled with a specific subset of optimizations. Interestingly, disabling certain optimizations can actually produce better performance as compared to keeping all optimizations enabled. Table 4.5 lists all the optimizations that don't cause any performance degradation for these benchmarks. For most of the Polybench and CPU SPEC 2017 benchmarks, the impact due to increased alias analysis precision on the performance of the benchmarks lies in the range of 0% to 10%. The occasional performance degradation can happen because optimizing the benchmarks aggressively can result in triggering wrong optimizations or an increase in code size which in turn degrades the performance of the benchmark.

In such cases, disabling specific optimizations during compilation can result in improving the performance of the benchmark.

4.4.3.1 Performance improvement potential of multi-threaded benchmarks

Although many of the optimizations enabled by Horizon are commonly discussed in the context of serial programs, they remain applicable to multi-threaded programs because they operate within the scope of individual threads. To assess the impact on the performance of multi-threaded programs using the program annotations inferred by Horizon+, we use the multi-threaded benchmarks of SPLASH-2 suite. We execute the benchmarks for different numbers of threads (1, 2, 4, 8, and 12), compiled with and without program annotations. Benchmarks `fft`, `radix`, `ocean_cp`, and `ocean_ncp` execute only when the number of threads is a power of 2. These benchmarks have properties that make the performance dependent on the numbers of threads [98].

Our primary goal is to estimate the upper bound on the improvement in alias analysis precision and its impact on performance. Although parallel programs can have different thread schedules and may contain data races, the optimizations enabled using precise alias information are still applied independently within each thread. Different schedules can potentially affect the profiling behavior and may lead to unstable executions. To handle such cases, our approach can perform profiling multiple times and generate an accumulated profile log across executions. However, such unstable behavior is rare in practice, and we did not observe it for the evaluated benchmarks. All the evaluated parallel benchmarks generated the expected output during the optimized executions. None of the benchmarks crashed or showed inconsistent behavior during profiling or optimized execution, and therefore rerunning the profiling phase to accommodate different schedules was not required.

Table 4.3 shows the performance impact (%) with different numbers of threads for each benchmark when compiled with 1) all optimizations enabled, and 2)

Table 4.3: Performance impact (AO = All optimizations enabled, MEO = The most effective combination of optimizations enabled. The values indicate percentages.)

	#Threads = 1		#Threads = 2		#Threads = 4		#Threads = 8		#Threads = 12	
Benchmark	**AO**	**MEO**	**AO**	**MEO**	**AO**	**MEO**	**AO**	**MEO**	**AO**	**MEO**
fft	-0.13	1.56	-0.21	0.45	-0.13	0.25	0.05	0.05	–	–
lu_cb	9.87	9.9	10.43	10.54	10.62	10.71	9.42	9.42	6.3	6.3
lu_ncb	8.48	8.75	8.43	8.85	7.18	7.44	5.11	5.58	5.78	6.18
radix	-0.06	0.06	0.44	0.72	0	0.3	0.18	0.69	–	–
barnes	5.83	6.09	5.13	5.13	7.64	7.64	4.87	5.53	6.23	6.23
ocean_cp	5.16	5.26	0.02	0.09	0.1	0.14	-0.02	0.02	–	–
ocean_ncp	0.42	0.69	0.92	1.17	0.58	0.58	-0.24	0.08	–	–
radiosity	16.97	17.98	15.99	17.36	14.93	15.68	7.51	10	7.25	8.7
raytrace	34.35	34.35	41.77	41.77	16.35	17.39	19.95	20.23	23.46	23.9
water-nsquared	3.44	4.41	11.74	12.73	9.94	10.61	11.14	11.59	9.73	10.57

the most effective combination of optimizations enabled. For multithreaded applications, compiler optimizations can further affect cache contention among threads. However, the overall performance of multithreaded executions is influenced by cache contention. To quantify this effect, we measure the number of HITM (loads that hit in a modified cache) events [99] using the perf tool [100]. A reduction in HITM events indicates a reduction in cache contention.

We observe that raytrace shows a significant improvement compared to the other benchmarks when compiled using program annotations. The main reason for this is a significant reduction in loads (22-30%), stores (23-25%), and HITM events (11-32%) for different numbers of threads. The reduction in HITM events (32%), loads (30%) and stores (25%) is the highest with two threads. The reduction in HITM events (11%) is the lowest for four threads.

For radiosity, the improvement is due to the reduction in loads (1-5%), stores (9-15%), and HITM events (-16-38%). The best performance improvement is achieved with one thread, and then it consistently decreases with the growing number of threads. This happens because as the number of threads increases, the reduction of HITM events (maximum for two threads, at 31.72%) decreases. For 12 threads, the number of HITM events is increased by 16%.

For water-nsquared, precise alias information leads to an approximate reduction of 50% in both loads and stores, irrespective of the number of threads, resulting in a good performance improvement. The performance gains are significantly higher when the number of threads is greater than one because of the huge reduction in HITM events in the range of 73% to 98%.

For ocean_cp, the number of loads decreases by approximately 10% for all threads. However, the number of HITM events increases by 18% for two threads and 4% for eight threads. For four threads, the number of HITM events decreases by 0.3%. For this benchmark, we don't observe significant improvement for multiple threads.

We observe that the benchmarks in Table 4.2, apart from the ones described above, show some difference in the performance impact when compiled with all optimizations enabled as opposed to when compiled with a specific subset of optimizations. Table 4.5 lists all the optimizations that don't cause any performance degradation for these benchmarks. It is not guaranteed that the most effective combination of optimizations remains the same as the number of threads increases. This is because increasing the number of threads affects cache behavior, which can further affect the effectiveness of optimizations.

Adding the 'restrict' keyword on function arguments based on Horizon+'s inferred annotations. We analyzed the impact of adding "restrict" to the function arguments based on Horizon+'s annotations. To safely insert "restrict", we identified the function arguments that were `no-alias` with all other pointers (not identified precisely by static analysis) based on the inferred annotations. At the IR level, we realize "restrict" by attaching LLVM's `NoAlias` attribute to the corresponding function arguments. It is worth noting that for all experiments, the annotations were incorporated after function inlining, reducing the number of function arguments available.

Table 4.4: Effectiveness of Horizon+ (Γ = # functions contributing to >95% of the execution time and α = the overall program performance improvement. τ_{pos} =# functions recommended for user-inspection with positive performance improvement per-function, and α_{pos} = the corresponding program performance improvement.).

Benchmark	Γ/α	τ_{pos}/α_{pos}
544.nab_r	4/1.02	3/1.02
538.imagick_r	4/1.17	1/1.4
505.mcf_r	9/3.65	5/3.65
557.xz_r	6/2.1	3/2.8
525.x264_r	146/11.56	23/12.88
502.gcc_r	846/3.82	154/3.82
500.perlbench_r	107/0.26	41/1.02
raytrace	6/34.35	3/34.34
radiosity	5/17.98	1/18.23

For all Polybench benchmarks, inlining reduced the code to a single remaining function argument, limiting the scope of meaningful evaluation. For four of the eight CPU SPEC 2017 benchmarks, the performance improvement was in the 0-1% range, with a maximum of 0.92% for `500.perlbench_r`. The remaining four benchmarks exhibited performance degradation, with a maximum of 1.12% for `557.xz_r`. For the multi-threaded SPLASH-2 benchmarks, only two out of ten benchmarks reported performance degradations with a maximum of ∼1.3% for `fft` (1 thread). Seven of the remaining eight benchmarks had performance improvements in the 0-10% range, with only `radiosity` showing an improvement of ∼18.7% for 1 thread.

4.4.4 Feedback to the user

RQ3 Can Horizon+ precisely identify functions for performance improvement when compiled using input-specific alias annotations?

The final phase of Horizon+ provides feedback to the user about functions that have reported high improvement within the chosen benchmarks. The information regarding the contribution of functions to execution times is obtained using Gprof [91]. The benchmarks are compiled with the "-pg" option, first using program annotations and then without them. The benchmarks are compiled with

the most effective set of optimizations identified after annotating the IR. As mentioned in section 4.3, the annotation inference phase infers annotations for the functions contributing to more than 95% (including system libraries) of the execution time when arranged in non-increasing order. We then analyze the profiled files generated using Gprof by executing the benchmarks. The functions with a positive improvement in the execution time are recommended to the user in non-increasing order of the improvement reported for user annotations for each benchmark.

Table 4.4 provides this information for the chosen benchmarks. The first column of the table lists the benchmark names. The second column presents the number of functions that contribute to more than 95% of the execution time along with the overall potential for performance improvement. The third column presents the number of functions recommended by Horizon+ for user annotations and the corresponding percentage performance potential improvement. The reduction in the number of functions recommended for user inspection ranges from 25% (`544.nab_r`) to 84% (`525.x264_r`). For the benchmarks (`525.x264_r`, `500.perlbench_r`, and `502.gcc_r`) where a significant number of functions contribute to more than 95% of the execution time, the reduction in the number of functions recommended for user inspection ranges from 62% to 84%.

We observed that the performance improvement potential is nearly equal to the optimal value α for five out of nine chosen benchmarks. For the remaining four benchmarks, the reported performance improvement potential is better than the optimal value α. As the recommended functions are the ones that report a positive improvement potential, no further optimization is performed for the functions which show a negative improvement potential (performance degradation). Thus, for such cases, the optimal performance improvement can be considered as α_{pos} instead of α. The users can wisely pick a threshold for the performance improvement potentials to limit the number of functions to annotate, according to their own requirements. For example, if users are concerned about

Table 4.5: Most effective combination of optimizations.

Benchmarks	Combination
2mm, nussinov	LICM
durbin, fdtd-2d, 500.perlbench_r	DSE
lu_ncb	GVN, DSE
gemm, radix	LICM, LV
ludcmp, heat-3d, ocean_cp	LICM, DSE
544.nab_r	LV, DSE
mvt	GVN, LV, SLP
gemver, gesummv, jacobi-2d, 502.gcc_r	GVN, LV, DSE
water-nsquared	GVN, SLP, DSE
3mm, bicg, adi, 538.imagick_r, 505.mcf_r	GVN, LICM, DSE
557.xz_r	LICM, LV, SLP
correlation, syr2k, trisolv, seidel-2d, fft	LICM, SLP, DSE
atax, jacobi-1d, lu_cb	GVN, LICM, LV, DSE
trmm, doitgen, 519.lbm_r, ocean_ncp	GVN, LICM, SLP, DSE
barnes	GVN, LV, SLP, DSE
covariance, gramschmidt, floyd-warshall, 525.x264_r, radiosity	LICM, LV, SLP, DSE
symm, syrk, cholesky, lu, deriche, raytrace	GVN, LICM, LV, SLP, DSE

safety, they can choose a high threshold value to minimize the trusted computing base. If they are concerned about performance, they can pick a smaller threshold value.

4.4.5 Optimization opportunities

RQ4 Which optimizations benefit the most from precise input-specific alias information for the given program?

Table 4.5 lists the optimizations which contribute in achieving the best performance for each benchmark, as reported in Table 4.2. We obtained this data by executing the benchmarks with different set of optimizations enabled. Among the optimizations, DSE and LICM appear most frequently, i.e. 42 and 36 out of the total 47 benchmarks respectively, whereas LV and SLP appears least frequently (24/47). All optimizations play a role in optimizing the four benchmarks for which we observed around 10% or more performance improvement (Table 4.2). In particular, DSE appears in all combinations for those benchmarks, whereas SLP appears only in one case. GVN, LV, and LICM each appear in three out of those four combinations. This trend continues across all

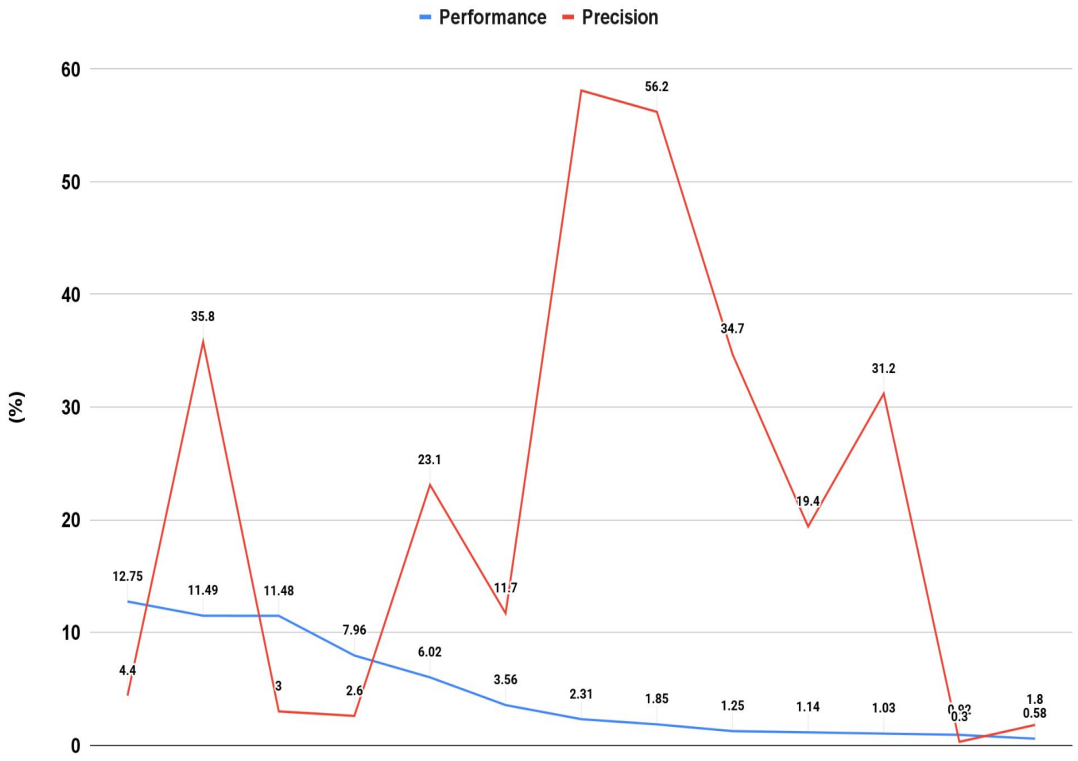


Figure 4.8: Per-function performance impact and precision improvement potential (x-axis represents the functions resulted in performance and precision improvement potentials of more than 0.5% and 0% respectively and y-axis represents the improvement potentials in percentage.).

benchmarks for which we observe that on average about four optimizations benefit from the availability of precise alias information. However, picking the right set of optimizations is tricky for multiple threads. A set of optimizations that works well for one thread might not be the best option for multiple threads, when cache contention also comes in the picture. For example, in `raytrace`, the set of optimizations that work for one and two threads is not the best option for other threads. The improvement for one and two threads is better when all optimizations are enabled (Table 4.3) whereas the other threads performs better when the optimization LICM is disabled.

4.4.6 Relationship between performance impact and precision improvement potential

The idea behind the technique mentioned in the paper is that certain optimization opportunities can be triggered if the precision of alias analysis is improved.

Table 4.6: Per-function performance vs precision improvement potentials for individual benchmarks (τ = Percentage increase in the total pointer pairs, α = Percentage increase in the no-alias and must-alias pointer pairs for the optimizations and PIP = Performance improvement potential (in %).).

Function	τ_{GVN}/α_{GVN}	$\tau_{LICM}/\alpha_{LICM}$	τ_{DSE}/α_{DSE}	τ_{LV}/α_{LV}	τ_{SLP}/α_{SLP}	PIP
GetVirtualPixelsFromNexus	41.65/45.4	0/0	0/0	0/0	0/0	12.75
next_nonempty_voxel	128.39/51.77	1293.62/28.7	260/57.23	0/0	187.5/75	11.49
compute_diff_disc_formfactor	64.91/9.13	0/0	25.6/0.96	0/0	1255.18/0	11.48
ConvertPrimRayJobToRayMsg	34.89/2.41	0/0	82.76/10.34	0/0	1250/0	7.96
get_mem4Dint	160/100	-33.34/100	0/50	616.67/23.26	0/0	6.02
next_nonempty_leaf	0/37.29	50/50	0/0	0/0	0/0	3.56
x264_pixel_satd_8x4	28.79/49.42	496.97/59.59	1200/80	0/0	0/91.42	2.31
lzma_decode	60.82/16.47	-10/20	0/21.43	0/0	0/0	1.85
lzma_mf_bt4_find	206.58/54.76	665.39/16.93	23.08/36.53	0/0	0/0	1.25
price_out_impl	68.04/46.74	230.29/17.26	104.32/-5.7	2333.34/0	-1.34/22.66	1.14
get_block_chroma	228.13/32.34	29.04/37.41	-8.34/46.97	0/0	0/0	1.03
S_find_byclass	474.54/3.63	46.02/18.74	46.02/18.74	0/0	0/0	0.92
Perl_regexec_flags	923.31/-12.3	38.94/32.54	38.94/32.54	0/0	0/0	0.58

Therefore, it follows naturally that improvement in precision must be correlated with improvement in the program's performance.

We tried to statistically determine the correlation between precision improvement potential and performance improvement potential for individual functions (contributing to more than 95% of the execution time of the benchmark) belonging to seven CPU SPEC 2017 and two SPLASH-2 benchmarks. The graph shown in Figure 4.8 plots the performance and precision improvement potentials for the functions that resulted in performance and precision improvement potentials of more than 0.5% and 0% respectively. The x-axis of the graph represents the function names and y-axis represents the improvement potentials in percentage. Surprisingly, the correlation turned out to be insignificant. In other words, there seems to be no direct dependence between the two factors.

We also investigated the per-function precision improvement potentials for individual optimizations relying on the alias analysis information. We computed the percentage increase in the number of pointer pairs generated by the optimizations and the percentage increase in the number of `no-alias` and `must-alias` pointer pairs due to the annotations for individual functions. Table 4.6 shows the data for individual optimizations for the functions reporting

more than 0.5% and 0% performance and precision improvement potentials. The number of pointer pairs generated by the optimizations might increase or decrease with respect to the native compilation when the annotations are used as it depends on the number of precise pointer pairs answered to enable the optimization. We observe a positive correlation between the percentage increase in the number of no-alias and must-alias pointer pairs for GVN optimization and the performance improvement of individual functions. However, there is no direct relationship observed between the estimates of precision and performance improvement potentials for other optimizations.

This is observed that not every pointer pair answered precisely contributes equally to the performance improvement of the program. Correct identification of the alias relationships of some pointer pairs might result in triggering effective optimizations, resulting in performance improvement. In other cases, the triggered optimizations might end up degrading the performance of the program. It is observed that precision improvement for alias/no-alias identification within a function scope doesn't necessarily improve the performance of the function. The conclusion that can be drawn is that the performance of a program depends on the pointer pairs which enable optimizations that significantly benefit the performance of the program. In most cases, we can expect optimizations to improve the performance of a program, and this is indeed reflected at a higher level when we look at the overall performance improvement potentials of the benchmarks.

4.5 Threats to Validity

The study mainly faces external threats to validity. The choice of benchmarks may influence the outcome of the study, if the benchmarks are not representative. However, to mitigate this threat, we have chosen benchmark suites that have been commonly used in recent related work. Similarly, the choice of the static analysis tool may influence the results. To counter this threat, we have used LLVM, which is a popular compiler and static analysis platform. The de-

fault alias analysis implementations as well as the optimizations provided by LLVM are state-of-the-art and highly scalable. Therefore, we believe that our results can be generalized to other static analysis tools and benchmarks.

Our approach also faces an internal threat to validity due to dynamically inferred non-aliasing annotations derived from a finite set of inputs. These annotations may not hold for all executions and are not intended to provide sound guarantees, which may lead to miscompilations if applied beyond the profiled inputs. While prior work on "soundy" program analyses [101] explores trading soundness for scalability and practical utility, their goal is to develop analyses that remain usable in practice despite limited guarantees. In contrast, our goal is fundamentally different, we do not aim to provide a generally applicable or deployable optimization technique, but rather to estimate the maximum achievable performance for a given input. We therefore expect users to profile with representative inputs and manually apply annotations only when correctness can be guaranteed.

4.6 Summary

This chapter presented a profiling-based approach to estimate the upper bound on the improvement potential when alias analysis precision is increased. The approach dynamically infers alias annotations by executing the program and incorporates them into the intermediate representation to expose missed optimization opportunities. These annotations serve as feedback to the user, highlighting functions that significantly contribute to performance improvements and identifying the most influenced optimizations. Based on this feedback, developers can choose to either improve the underlying alias analysis implementation or selectively annotate the program to enable critical optimizations. Our evaluation showed that the precision improvement potential of LLVM's default alias analysis ranged from 0% to 33%, while the corresponding performance gains reached up to 53% for the provided input test cases. These results reveal considerable scope for improvement, particularly by enhancing interprocedural alias informa-

tion. Notably, several benchmarks saw significant benefits from loop-level optimizations enabled by more precise alias information.

Future work includes incorporating interprocedural alias information in a scalable manner to further enhance precision without compromising performance. This could enable additional loop-level optimizations that are currently missed due to conservative analysis. Another promising direction is to extend the approach for input-agnostic annotation inference.

Chapter 5

An Alignment-Based Allocator

Static alias analysis in production compilers is typically intra-procedural and conservative, limiting its ability to disambiguate pointers across function boundaries. As a result, compilers often miss optimization opportunities. Existing techniques [24, 25, 102] and production compilers like GCC and LLVM [26, 27] implement loop-versioning with dynamic checks for overlapping memory accesses within a loop, due to the limitations of static alias analysis. The loop version without overlapping accesses can be further optimized by the loop transformation passes.

The polyhedral model [71–73] and symbolic range analysis [74–76] can compute an $O(1)$ dynamic range check to disambiguate the memory regions accessed within a loop. These checks are placed outside the loop and used as a condition for loop-versioning. These techniques can disambiguate memory regions even if they belong to the same array. The polyhedral analysis requires loop bounds and all data access functions to be affine combinations of loop induction variables and loop invariants [24, 71]. Symbolic range analysis can additionally handle non-affine accesses within a loop; however, polyhedral analysis is more precise when both techniques can be applied to a loop [24].

Both polyhedral analysis and symbolic range analysis require:

1. Loop bounds to be loop invariants, and
2. Loop that iterates using an affine induction variable.

The existing implementation of the LLVM compiler also uses loop-versioning and $O(1)$ dynamic checks to improve program performance by disambiguating pointers at runtime. However, it also requires loops to satisfy the conditions mentioned above. The LLVM compiler uses scalar evolution analysis [77, 78] to compute the loop bounds and affine memory accesses within a loop.

Consider the following example:

```
void foo(int *a, int *b, int *c, int *size) {
  for (int i = 0; i < *size; i++)
      a[i] = b[i] + c[i];
}
```

In the function `foo`, the upper bound of the loop (i.e., `*size`) is not a loop invariant, because it can overlap with the array `a`, which is being updated in the loop. Thus, neither polyhedral analysis nor symbolic range analysis can insert a dynamic check to disambiguate the array accesses within the loop.

For such loops, Alves et al. [24] propose another technique, namely a purely dynamic analysis that uses dynamic checks to infer if the memory accesses are performed on different memory allocations. If so, they safely conclude that the memory accesses do not overlap, assuming that the behavior of out-of-bounds object accesses is undefined. Thus, this technique does not add any constraint on the array indices used to access the memory inside the loop.

The key idea in this scheme is to attach a unique tag to every object during allocation. At runtime, a red-black tree lookup is used to compute the starting address of the object from an internal address. The starting address also serves

as a unique tag for the object. Two pointer addresses never overlap if the starting addresses of the corresponding objects (or tags) are different.

The cost of these checks is logarithmic in terms of the number of live memory allocations ($O(log\ n)$ where n represents the number of live memory allocations). Such high overheads for dynamic checks make it unsuitable for large applications. Our work is motivated by the purely dynamic approach suggested by Alves et al. [24]. However, our focus is on minimizing the overheads introduced by the dynamic checks that would make the approach practical for real-world applications.

We would also like to argue that such code patterns, in which the loop bounds are not loop invariants, are not uncommon. In Figure 5.6, we have listed some code patterns from the CPU SPEC 2017 benchmark suite that show more than 20% improvement with our technique. Thus, optimizing these loops is a significant problem that needs to be addressed.

This work focuses on reducing the overhead of dynamic checks for loops on which polyhedral or symbolic range analysis cannot be applied. Our approach, encapsulated in a tool called SCOUT, performs dynamic checks for these loops in $O(1)$. The key idea is to constrain the size and alignment during memory allocations so that SCOUT can disambiguate two pointers in just one memory access. Our approach is inspired by classic buddy allocators [103], which enforce size and alignment-based organization of memory through efficient splitting and merging of objects. Similar design principles have also been employed in memory safety systems such as Baggy Bounds Checking [104], where aligned, size-controlled regions are used to enforce spatial safety. In contrast, we leverage these ideas solely for fast pointer disambiguation rather than for ensuring memory safety. This design choice also enables very efficient runtime checks. The overheads of our checks are considerably lower than the $O(logn)$ [24].

SCOUT implements loop-versioning for an already optimized loop, with runtime checks for disambiguation of arrays accessed within the loop. The loop transformations are performed on the loop version in which the array accesses do not overlap. If the additional non-overlapping information actually enables some optimization, SCOUT keeps both versions of the loop; otherwise, it rolls back to the original code. At runtime, if a disambiguation check fails, execution transfers to the original unoptimized loop version. Furthermore, dynamic checks for disambiguation are added only for those array accesses that are needed to enable the loop transformation. SCOUT statically analyzes the transformed and original loops to identify the checks required for the transformation.

The potential benefits of our loop-versioning algorithm cannot be inferred solely at compile time because they further depend on the path taken at runtime and the number of loop iterations. Additionally, SCOUT uses a runtime profiler to identify loops for which the cost of dynamic checks is higher than the benefits of loop transformation. SCOUT disables loop-versioning for such loops.

We integrated SCOUT with LLVM by implementing a new pass in the LLVM compilation infrastructure. We evaluated our technique using the Polybench and CPU SPEC 2017 benchmark suites.

This work makes the following contributions:

1. An efficient technique that allows the compiler to explore more optimization opportunities based on alias analysis. Our novel technique combines loop-versioning with constant-time dynamic checks, explained in Section 5.2, to disambiguate memory accesses. The dynamic checks are performed using a custom allocator, which is presented in Section 5.2.2. The technique further uses a feedback mechanism to reduce the number of dynamic checks, explained in Section 5.2.
2. Implementation and integration of our technique with LLVM.

3. Estimation of the performance benefits for the Polybench and CPU SPEC 2017 benchmark suites.

5.1 Motivating example

In this section, we discuss an example to motivate our technique. Consider the code snippet shown in Figure 5.1a, which adds the elements of arrays `b` and `c`, storing the result in array `a`. The compiler can vectorize this loop if it knows for certain that `b`, `c`, and `size` do not overlap with `a`. Line 10 in Figure 5.1b shows the vectorized loop body, where ``` ``i:i+3'' ``` denotes four operations at indices `i, i+1, i+2, i+3` performed in parallel, replacing the single scalar operation at line 17. If array `a` overlaps with `b`, `c`, or `size`, parallel execution may yield incorrect results. In this case, the alias relationships depend on the values passed by the caller and cannot be resolved through intraprocedural alias analysis.

As discussed earlier, polyhedral, symbolic range, and scalar evolution analyses require the loop bounds to be loop invariants, which is not true in this example. Therefore, in addition to SCOUT, the pointer disambiguation technique in Alves et al. [24] can be used to version this loop.

SCOUT inserts dynamic checks to rule out the possibility of overlapping between `a, b, c` and `size`. The dynamic checks ensure that memory accesses using `a, b, c` and `size` belong to different objects, and thus they cannot overlap no matter what the value of `i` is at runtime. For fast dynamic checks, SCOUT sets the allocation size and the alignment of an object during the allocation to the nearest 2^K, where 2^K `>= allocation size`. This design can introduce additional memory overhead due to internal fragmentation caused by rounding allocation sizes to the nearest 2^K; however, this tradeoff enables fast constant-time dynamic checks. The objects are allocated from segments. All objects on a segment are of the same size, which is a power of two. The starting address of a segment is always aligned to 4GB. The object size for a segment

```
void foo(int *a, int *b,
    int *c, int *size) {
1. for(int i = 0;
     i < *size; i++)
2.  a[i] = b[i] + c[i];
}
```

(a) The original code.

```
void foo(int *a, int *b,
    int *c, int *size) {
3. size_t s
    = GetObjectSize(a);
4. if(IsNoAlias(a, b, s) &&
5.    IsNoAlias(a, c, s) &&
6.    IsNoAlias(a, size, s)) {
7.   int t = *size;
8.   int i;
9.   for (i = 0; i+3 < t;
        i = i+4)
10.   a[i:i+3]
        = b[i:i+3] + c[i:i+3];
11.
12.  for (; i < t; i++)
13.   a[i] = b[i] + c[i];
14. }
15. else {
16.  for(int i = 0;
       i < *size; i++)
17.   a[i] = b[i] + c[i];
18.  }
}
```

(b) The transformed code using SCOUT.

Figure 5.1: An example to demonstrate the overview of SCOUT. SCOUT inserts dynamic checks to rule out the possibility of overlapping between `a`, `b`, `c` and `size`. The compiler could vectorize the loop in the scope in which memory accesses do not overlap.

is stored on the top of the segment. We discuss the structure of a segment in more detail in Section 5.2.2 using Figure 5.3. The starting address of a segment can be computed using just an "`and`" operation from any internal address of the segment using the alignment property of the segment. Two objects can overlap if and only if they belong to the same segment. We can check if two objects on a segment overlap using an "`xor`" operation and compare it with the object size of the segment. To check non-overlapping (non-aliasing) of two pointer addresses, we first need to compute the object size of the first pointer (at line-3). The `GetObjectSize` routine uses the alignment property of the segment to compute the size in just one memory access. Once we know the size, it is passed to the `IsNoAlias` routine, which takes two pointers and the

object size of first pointer as input and returns true if the pointers point to different objects. This routine does not access any memory.

SCOUT adds dynamic alias checks for every read-write and write-write pointer pair because the static alias analysis returns *may-alias* for each of these queries. SCOUT inserts additional metadata in the if-block such that the compiler can treat each of these pointer pairs (in the if-block) as *no-alias* during the transformation passes. After this, SCOUT performs loop transformation optimizations for the loop in the if-block. With the additional non-aliasing information, the compiler can now vectorize the loop in the if-block, as shown in Figure 5.1b. At line-10, the syntax "i:i+3" is used to denote that four parallel writes at indices i, i+1, i+2, i+3 are performed using a single instruction. If the compiler cannot transform the loop with the additional non-aliasing information in the if-block, SCOUT restores the original code.

Unlike LV, other optimizations, i.e., LICM, DSE, and GVN, do not require checks for every read-write and write-write pair. For example, if a load `LD` is moved outside the loop, we need to check that `LD` does not overlap with the writes present in the loop. Let us consider that instead of vectorizing the loop, the loop transformation pass in Figure 5.1b only moves `*size` outside the loop (at line-7). In this case, we need to check that ``size'' and ``a'' do not overlap instead of adding three checks as in the case of vectorization. SCOUT statically analyzes the original and transformed loop in the if-block and adds only those checks needed for the transformation.

Finally, it is possible that the optimized path is rarely taken at runtime, or the overhead of the runtime checks is more than the benefit of the loop transformation. SCOUT uses a profiler to disable loop-versioning for such loops.

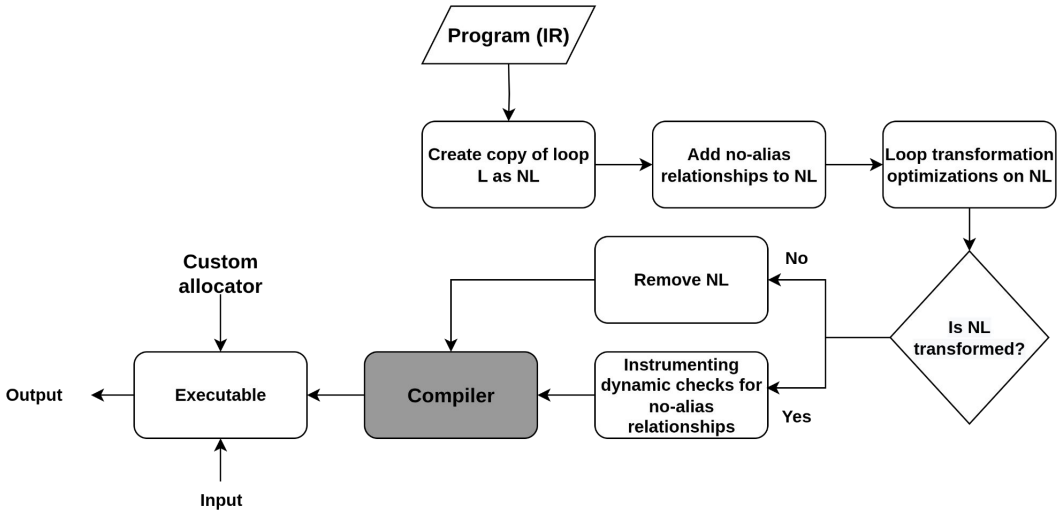


Figure 5.2: Architecture of SCOUT.

5.2 SCOUT

Figure 5.2 shows the architecture of SCOUT. SCOUT works on the input program's intermediate representation (IR). It performs versioning for an already optimized loop with checks for non-aliasing (non-overlapping) memory accesses within the loop. Loop transformation optimizations are performed on the non-aliasing version of the loop. If the loop is not transformed, it is discarded; otherwise, dynamic checks to detect non-aliasing memory accesses are inserted into the program. The transformed program is sent to the compiler to generate the final executable. At runtime, a custom memory allocator is linked to the executable generated by SCOUT. We will now discuss our approach in detail in the rest of this section.

5.2.1 Loop-versioning

Algorithm 1 shows the steps followed by SCOUT to perform loop-versioning. The algorithm takes a function IR `F` and a loop `L` as arguments. This algorithm is performed after all loop transformations except vectorization have been performed on L. The reason is that it is hard to optimize an already vectorized loop with additional non-overlapping information. However, the additional non-

aliasing information enables further transformations in a non-vectorized loop. Vectorization is later performed on the transformed loop.

```
   Input: Function F, Loop L
   Result: Transformed/Untransformed Loop
 1 begin
 2     IsVectorizableBefore ← isVectorizable(L);
 3     NL ← cloneLoopWithDummyCheck(L);
 4     insertNoAliasRelation(NL);
 5     RunLICM(NL);
 6     RunGVN(F);
 7     RunDSE(F);
 8     IsVectorizableAfter ← isVectorizable(NL);
 9     if isLoopTransformed(L,NL) or
         (not IsVectorizableBefore and IsVectorizableAfter) then
10         identifyDynamicChecks(L,NL);
11         insertDynamicChecks(L,NL);
12         removeNoAliasRelation(NL);
13     else
14         removeLoopAndDummyCheck(NL);
15     end
16 end
```

Algorithm 1: `Loop-versioning` algorithm of SCOUT.

At line-3, the algorithm creates a copy of the original loop and inserts a dummy check for non-aliasing. We refer to the versioned loop as `NL` in this section. At line-4, it inserts `no-alias` metadata in the non-overlapping version of the loop. The compiler can use `no-alias` metadata to infer that two memory accesses do not overlap. The `no-alias` metadata is added for only those pointer pairs whose bases are loop invariants, and it is safe to access at least one of them inside the loop's preheader. It then performs LICM, GVN, and DSE optimizations (from line-5 to line-7) on the non-overlapping version of the loop (i.e., `NL`). The `isVectorizable` routine at line-2 and line-8 does not vectorize the loop. It only checks that, at this point, the compiler can vectorize this loop if needed.

SCOUT takes feedback from static optimizations and decides whether the loop was optimized or not. The versioned loop (`NL`) is considered to be optimized if:

1. Any of the loop transformation optimizations has transformed `NL` or,
2. `L` cannot be vectorized, but `NL` can be vectorized.

If the loop transformation optimizations do not optimize `NL`, then `NL` and the dummy checks are removed at line-14. Otherwise, actual dynamic checks are inserted to check the non-overlapping of memory accesses within the loop. Based on the feedback from static optimizations, SCOUT identifies the dynamic checks that need to be inserted to check the non-overlapping of memory accesses within the loop. These dynamic checks are chosen based on the following cases:

Case 1: If `L` is not vectorizable, but `NL` can be vectorized, insert dynamic checks for the base pairs of all possible pairs of the read-write and write-write memory accesses present in the loop body.

Case 2: If case-1 is not true, for every read memory access R, which is either moved outside of the loop body or removed from the loop body, insert dynamic checks for the base pairs of R and all write memory accesses present in the loop body.

Case 3: If case-1 is not true, for every write memory access W, which is either moved outside of the loop body or removed from the loop body, insert dynamic checks for the base pairs of W and all other (read/write) memory accesses present in the loop body.

For the identified base pointer pairs, the dynamic checks (Section 5.2.3) are instrumented (at line-11). At line-12, the `loop-versioning` algorithm removes `no-alias` metadata for pointer pairs for which the dynamic checks were not added in the previous step.

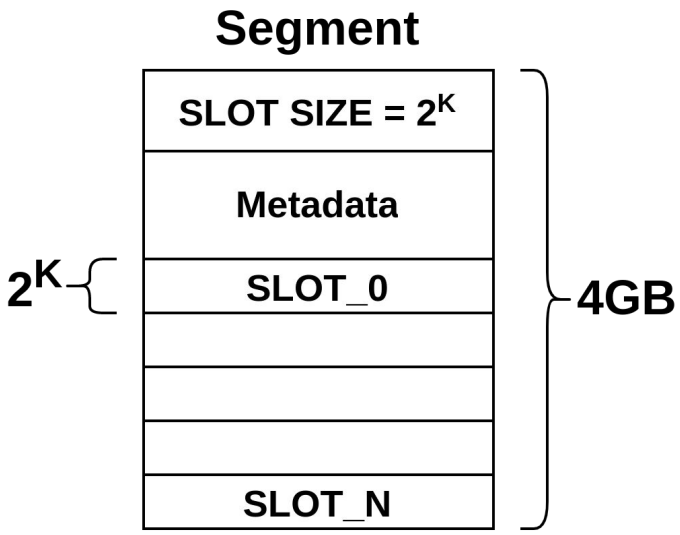


Figure 5.3: Structure of a Segment.

5.2.2 Custom Allocator

The allocator maintains a list of segments. A segment is a 4GB (configurable at compile time) contiguous virtual address space as shown in Figure 5.3. The starting address of a segment is a 4GB aligned address. The segment is further divided into fixed-size slots. The size of the slot is 2^K, where K is fixed for a given segment. The starting address of the slot is aligned to 2^K. Objects of different sizes are allocated from different segments. A slot is returned to the caller for each allocation from a segment. The first few slots of a segment are reserved for the metadata. Metadata includes a bitmap to keep track of free slots—the first eight bytes of a segment store the size of the slots of that segment. At runtime, SCOUT reads the first eight bytes in the segment header to determine the object's size.

5.2.3 Dynamic Checks for Non-overlapping

The dynamic check for the non-overlapping property for a pointer pair (P1, P2) is performed in two steps. In the first step, SCOUT computes the size of pointer P1, which involves memory access. In the second step, SCOUT checks the non-overlapping of P1 and P2 without memory access.

```
#define SEGMENT_SIZE (1ULL << 32)
size_t GetObjectSize(void *Ptr) {
    Segment *S = (Segment*)((size_t)Ptr & ~(SEGMENT_SIZE-1));
    if (S == DATA_SECTION_SEGMENT)
        return MAX_VIRTUAL_ADDR;
    return S->SlotSize;
}

bool IsNoAlias(void *P1, void *P2, size_t Size) {
    return xor((size_t)P1, (size_t)P2) >= Size;
}
```

Figure 5.4: Dynamic check to determine if a pair of pointers do not overlap. The `GetObjectSize` routine returns the object size from a pointer address. It uses the alignment property of the segment to read the size of the slot that is stored on the first eight bytes of a segment. The `IsNoAlias` routine takes two pointer arguments and the object size of one of the arguments and returns true if the input pointers point to different objects.

Size computation. The size computation is done using `GetObjectSize` in Figure 5.4. SCOUT computes the address of the segment by resetting the last 32-bits of the address (note that the starting address of a segment is aligned to 4GB). If the segment address is equal to the segment corresponding to the data section, then the pointer is a global variable. In this case, `GetObjectSize` returns the maximum virtual address available on the host platform. Notice that the global variables are not allocated from the segment, and thus we cannot compute their sizes at runtime. The size computation logic for stack allocation is discussed in Section 5.3. The first eight bytes on the segment store the size of the slot (Section 5.2.2), represented using the `SlotSize` field in the pseudo-code. The size of the slot is the size of the object, which is returned to the caller.

Check for non-overlapping. The `IsNoAlias` routine in Figure 5.4 takes two pointers and the size of one of these pointers as an argument. It returns true if the pointer arguments belong to different objects. Because of the property of a segment, if two objects are of a different size, they belong to different segments. In this case, the ``xor'' of two addresses would be more than or equal to the size of the segment. Two objects can only overlap if they

belong to the same segment; in that case, if the ``xor'' of two addresses is equal to or greater than the size of the slot, they cannot overlap (because of the alignment property of the slots). However, being in the same segment does not imply aliasing. The objects are still placed in distinct, non-overlapping slots unless their address ranges intersect. Notice that in the case of global variables, the object size is equal to the maximum virtual address on a 64-bit platform (as returned by the `GetObjectSize` routine). Due to this, `IsNoAlias` considers two global variables as may-aliases.

5.2.4 Profiler

The performance benefits of our approach can only be determined at the time of execution as they depend on the number of times the optimized loops are executed. One way of obtaining this information is by executing the program with a profiler. This leads us to the implementation of a profiler for SCOUT to maximize the benefits. This profiler identifies loops that improve the program's overall performance by executing them.

In the presence of the profiler, SCOUT works in two phases. In the first phase, the program is instrumented to collect the execution times of loops using the *rdtsc* instruction [105] by executing it on a given set of inputs for both versions of the loop. In the second phase, SCOUT uses the generated profile files to identify the transformed loops that improved the program's overall performance. These loops are identified based on different thresholds such as 10, 20, and 30, representing the percentage improvement in the execution time of the loop. SCOUT then performs versioning only for these loops.

5.3 Implementation

We implemented SCOUT as a part of the LLVM infrastructure (v10.0.0). We have added a pass to LLVM that follows the approach discussed in Section 5.2. We have integrated our pass with the Clang `O3` optimization level, and it runs by default with the `O3` option. The pass can be disabled using the option

disable-additional-vectorize. We extended the widely used production allocator `JEMALLOC-5.2.1` to implement our segment-based allocator.

JEMALLOC. `JEMALLOC` is a high-performance memory allocator designed for scalability and low fragmentation, widely adopted in multi-threaded and memory-intensive applications. It functions primarily as a *buddy allocator*, organizing memory into `size-based buckets` to efficiently handle allocation requests of varying sizes. For each bucket size, `JEMALLOC` allocates a large contiguous memory region, called an `extent`, using the `mmap` API, and divides it into fixed-size slots. These slots are cached and served during allocation to reduce overhead and latency. To minimize contention and enhance parallel performance, `JEMALLOC` uses per-thread caches, allowing threads to manage memory independently without frequent global synchronization.

We leveraged the caching framework of `JEMALLOC` and modified the extent allocation logic to use the segments (Section 5.2.2) instead of `mmap`. For an extent that is used to serve the objects of size 2^K, we allocate the extent from a segment that is used to allocate objects of size 2^K. We also ensure that an extent is not recycled for a different bucket size.

Extending the size to 2^K may create fragmentation issues, especially for large objects. `JEMALLOC` uses a bucket-based allocation strategy for small objects; because of that, the additional fragmentation caused by our technique is reduced. We found that the bucket (or class) sizes used by the `JEMALLOC` are not always 2^K. The bucket sizes used by `JEMALLOC` in bytes are 8, 16, 32, 48, 64, 80, 96, 112, 128, 160, and so on. For large objects, our fragmentation issue could be severe. For example, imagine a scenario where we need to allocate 2GB of memory for an allocation request for the size “1GB + 1Byte”. To mitigate this issue, for object size larger than 2^{14}, we map physical pages that are actually needed for the allocation. For example, in the case of “1GB + 1Byte”, we map the physical pages corresponding to “1GB + 4096Bytes”,

where 4096 is the page size. The virtual address reserved for a memory allocation is always 2^K. The average memory overhead of our scheme for CPU SPEC 2017 benchmarks is 7.47%, as discussed in Section 5.4.

An alias query can also be made for the stack objects at runtime. In most cases, the compiler can statically tell whether a stack object does not alias with another object. However, if a stack address escapes from the static scope, i.e., 1) stored in memory, 2) passed to a routine, or 3) cast to an integer, the compiler may not identify the stack object uniquely. For these cases, we tried replacing the stack allocations with `malloc` and `free`. However, the overhead of this approach is very high. To reduce the overheads, we used custom per-thread bump allocators for bucket sizes (in bytes) 8, 16, 32, ..., and 1024. In this setting, for objects larger than 1024, `malloc` and `free` are used. The bump allocators use segments with slot sizes equal to the bucket sizes of the bump allocators. Even with the bump allocators, the overheads are high for some benchmarks. To reduce the overheads further, we switch to a new stack during the main routine. The new stack is allocated from a segment for which the slot size is 64. If the size of an escaping stack object `X` is less or equal to 64, we set the alignment of `X` to 64. Due to this, at runtime, if the size of `X` is queried using the `GetObjectSize` (see Section 5.2.3) API, it will correctly return 64. If the object size is more than 64, we use per-thread bump allocators as discussed before. For multi-threading, SCOUT inserts a custom wrapper around calls to `pthread_create`. In the wrapper code, it allocates a 64-bit aligned stack from the segment and uses `pthread_attr_setstack` to set the new stack for the target thread. We discuss the overheads of our allocator in detail in Section 5.4.

Some library functions, such as `__ctype_b_loc`, `__ctype_toupper_loc`, etc., return its internal addresses that may not belong to a valid segment. SCOUT adds wrappers for these routines. The wrappers allocate a new object, copy contents from the library's object to the new object, and return the allocated

address to the caller. If the contents of libraries internal object never change (e.g., an internal array is used by the library for character classification functions such as `isalpha`, `isspace`, etc. that are not updated post-initialization), the new object is cached and reused during the subsequent use of the library function.

Embedding non-aliasing information. SCOUT inserts the non-aliasing information to the loop body with non-overlapping memory accesses. The compiler uses the non-aliasing information to check if two memory accesses do not alias statically. This information is incorporated using the `alias.scope` and `noalias` metadata of LLVM [93]. It specifies that the two pointers do not alias each other. This metadata can only be attached to memory instructions like load and store.

Because the array bases corresponding to memory accesses in the loop may not necessarily be memory instructions, SCOUT inserts a custom intrinsic [106] to the program to specify the `no-alias` relationships between the base pairs of the pointers. Our custom LLVM intrinsic is known as `llvm.custom.noalias`. This intrinsic holds a pair of pointers as arguments. The pair of pointers present in the custom intrinsic are treated as `no-alias` with each other. These pointer pairs are wrapped in the form of `LLVM's metadata nodes`. Sample usage of the intrinsic is shown below:

```
call void @llvm.custom.noalias(metadata DATA_TYPE %a,
metadata DATA_TYPE %b)
```

where `metadata` represents the LLVM's metadata node, `DATA_TYPE` represents the type of the pointer, %a and %b represent base pointers.

We also modified the static alias analysis algorithm to use our intrinsic. Suppose the original static alias algorithm cannot find the alias relationship between a pointer pair (P1, P2) at a given point. In that case, our modified algorithm additionally checks the presence of our custom intrinsic in the current or an outer scope with bases of `P1` and `P2` as operands. If it exists, the alias analy-

sis algorithm treats the pointer pair as `no-alias`.

Profiler. The profiler implemented in SCOUT generates two types of profile files in the first phase. The first file is generated to obtain the time taken statistics for loops when the program is executed, disabling our approach using the options *execute-unoptimized-path* and *get-time-stats* together. The second file is generated to obtain the time taken statistics for loops when the program is executed with our approach using the option *get-time-stats*. The profiler uses the generated profile files in the second phase to identify loops that benefited the program's overall performance for different thresholds. This can be done by specifying the options *allow-ben-loops* and *ben-loop-threshold* together. The user can specify different thresholds (in percentage) using the option *ben-loop-threshold*. The default value of the threshold is set to 0%.

Optimizing dynamic checks. SCOUT implements loop-versioning algorithm (Algorithm 1) for the innermost loops, which is consistent with the LLVM policy of vectorizing only the innermost loops. SCOUT moves the logic to generate the condition for the dynamic check to the preheader of an outer loop if it is safe to access the base address (for obtaining the size) in the preheader of the outer loop. To identify if a base pointer is safe to access at point P, SCOUT statically checks if a pointer derived from the base pointer is guaranteed to be accessed if the execution reaches P.

5.4 Evaluation

We evaluated SCOUT using 30 Polybench (version 4.2.1) and 16 C and C++ benchmarks available as a part of Intrate and FPrate suites CPU SPEC 2017 [95, 107] benchmarks. For Polybench, we used the default input set, and for CPU SPEC 2017, we used the reference input set. We compiled all the benchmarks with the `O3` optimization level. We performed the experiments on a 3.60GHz Intel(R) Core(TM) i9-9900K CPU 8 core machine with an x86_64 architecture and 32 GB primary memory, which uses a 64-bit Ubuntu 20.04.2

LTS operating system. We disabled hyper-threading during our experiments. We considered the arithmetic means of the execution time for five runs of each benchmark. We also report geometric means. The geometric mean is computed using multiplicative scaling factors instead of signed percentages. Hence, overhead values are first converted to factors (e.g., 8% = 1.08, -10% = 0.9), the geometric mean is computed on these factors, and the result is converted back to percentage form.

5.4.1 Memory and CPU Time Overhead of the Custom Allocator

Table 5.1 shows the memory and CPU time overheads of the CPU SPEC 2017 benchmarks. Column-2 and Column-5 show the memory and CPU overhead of the custom allocator with respect to the native `JEMALLOC` allocator, respectively. Column-3 and Column-6 show the memory and CPU overhead of the bump allocator with respect to the native `JEMALLOC` allocator. Column-4 and Column-7 show the memory and CPU overheads of replacing stack allocations with calls to `malloc` and `free` when the stack objects go out of scope with respect to the native `JEMALLOC` allocator.

We used the "Maximum resident set size" reported by the "/usr/bin/time -v" command for memory overheads. The memory overhead of our custom allocator is in the range -2.29% to 34.89% for CPU SPEC 2017 benchmarks (Table 5.1, column-2). The memory overhead of `povray_r` is high, as the peak memory consumption for this benchmark is only 8.4 MB. The major memory overhead comes from our custom stack and the page-table pages corresponding to segments. For other benchmarks, the memory overhead is always less than 20%. The geometric mean of memory overhead is 7.47%.

The CPU time overhead of our modified allocator is in the range of -2.57% to 8.36% for CPU SPEC 2017 benchmarks (Table 5.1, column-5). The percentage change in the execution time of the custom allocator w.r.t. the native allocator represents the CPU time overhead. In our design, the objects (an extent in the

Table 5.1: Memory and CPU overhead for CPU SPEC 2017 benchmarks. (MC = Memory overhead of using custom allocator w.r.t. to native `JEMALLOC` allocator, MB = Memory overhead of using bump allocator w.r.t. to native `JEMALLOC` allocator, MM = Memory overhead of replacing stack alloactions with calls to `malloc` and `free` when the stack objects go out of scope w.r.t. to native `JEMALLOC` allocator, CC = CPU overhead of using custom allocator w.r.t. to native `JEMALLOC` allocator, CB = CPU overhead of using bump allocator w.r.t. to native `JEMALLOC` allocator, CM = CPU overhead replacing stack alloactions with calls to `malloc` and `free` when the stack objects go out of scope w.r.t. to native `JEMALLOC` allocator. The values indicate percentages.)

	MO			CO		
Benchmark	**MC**	**MB**	**MM**	**CC**	**CB**	**CM**
namd_r	-2.29	-2.22	-2.09	0	0.51	0.51
parest_r	11.72	11.79	8.06	0.43	1.15	1.51
lbm_r	0.02	0.03	0.02	-0.61	-0.25	-0.25
imagick_r	-0.33	-0.32	-0.28	1.4	0.37	29.98
nab_r	17.58	17.6	17.56	1.54	0.73	0.55
perlbench_r	6.53	6.46	5.98	0.33	8.55	594.75
gcc_r	6.15	5.67	3.72	-0.98	1.22	168.86
mcf_r	0.01	0.01	0.03	-2.15	-2.07	-1.61
omnetpp_r	16.72	16.81	16.79	8.36	14.08	212.29
xalancbmk_r	19.97	19.99	19.99	3.05	3.36	189.64
x264_r	0.06	0.08	0.04	-2.36	-1.93	22.46
deepsjeng_r	0.08	0.06	0.08	1.64	2.41	77.89
leela_r	10.02	9.89	10.59	1.31	3.56	67.86
xz_r	0.05	0.06	0.06	0.61	4.43	4.36
povray_r	34.89	36.79	33.18	7.5	11.87	378.53
blender_r	5.2	5.25	5.21	-2.57	1.78	55.82
GM	**7.47**	**7.54**	**7.03**	**1.05**	**3.01**	**72.75**

case of `JEMALLOC`) from buckets of large sizes cannot be recycled for buckets of smaller sizes. But this is not true for the unmodified allocator. Because of this, the behavior of the per-thread caches is different in both runs. Using better allocation tuning can minimize this problem. We plan to investigate this in the future. Nevertheless, only three out of the 16 benchmarks incur CPU time overheads above 3%. With the usage of custom per-thread bump allocators (discussed in Section 5.3), the CPU overhead for three benchmarks (LoC $>$ 300K), namely, `perlbench_r`, `gcc_r`, and `parest_r`, is either negative or less. The geometric mean of CPU time overhead is 1.05%. The CPU overhead lies in the range of -2.07% to 14.08%, when we use bump allocator instead of

Table 5.2: Memory and CPU time overhead for Polybench benchmarks. We report the benchmarks that resulted in the absolute CPU overhead of more than 1%. (MD = Memory overhead of custom allocator w.r.t. native `JEMALLOC` allocator when compiled using the default size of memory allocations, MA = Memory overhead of custom allocator w.r.t. native `JEMALLOC` allocator when the size of the memory allocations is aligned to 2^K, CD = CPU overhead of custom allocator w.r.t. native `JEMALLOC` allocator when compiled using the default size of memory allocations, CA = CPU overhead of custom allocator w.r.t. native `JEMALLOC` allocator when the size of the memory allocations is aligned to 2^K. The values indicate percentages.)

	MO		CO	
Benchmark	**MD**	**MA**	**CD**	**CA**
gemm	0.1	0.4	5.3	0.02
symm	-0.04	0.33	2.01	1.22
doitgen	0.26	0.41	-2.55	0.04
jacobi-2d	0.17	0.7	-1.45	0.01

custom stack (Table 5.1, column-6), as discussed in Section 5.3. However, the CPU overhead of replacing stack allocations with calls to `malloc` and `free` is considerably high for these benchmarks (Table 5.1, column-7). To reduce this overhead, we use the bump allocator and the custom stack as discussed in Section 5.3.

Table 5.2 presents the memory and CPU time overheads for the Polybench benchmarks for which the absolute CPU overhead is more than 1%. The memory overhead of our custom allocator lies in the range of -0.41% to 0.72% with a geometric mean of 0.13%. The CPU overhead of our custom allocator lies in the range of -2.55% to 5.3%, with a geometric mean of 0.21%.

In the case of `gemm`, `doitgen`, and `jacobi-2d`, our custom allocator either performs better or worsens the performance as compared to the native allocator. This is due to the fact that our custom allocator aligns the memory allocations to 2^K. To verify this, we performed another experiment, in which we align the sizes of the memory allocations to 2^K in the source code itself. We observed negligible overheads in that case for these three benchmarks. However, we observed a CPU time overhead of 1.22% (Table 5.2, column-5) for `symm`

Table 5.3: Performance benefits for Polybench without profiler. We report six benchmarks that resulted in performance benefits of more than 3%. (#L_a = Total number of versioned loops by SCOUT, P_a = Performance benefits when compiled with SCOUT, P_r = Performance benefits when compiled with the `restrict` keyword, V_c = Variance of the execution time when compiled with the custom allocator, V_s = Variance of the execution time when compiled with the custom allocator and SCOUT. Performance-benefit numbers represent percentages.)

Benchmark	#L_a	P_a	P_r	V_c	V_s
gesummv	1	49.37	48.42	0.00000001	0.00000001
2mm	2	4.22	4.1	0.00025563	0.00015497
3mm	3	4.28	4.03	0.00017649	0.00009542
bicg	1	51.11	51.01	0.00000001	0.00000001
doitgen	2	10.2	11.11	0.00000132	0.00000334
jacobi-1d	2	11.8	-1.11	0.00000001	0.00000001

even when the sizes are aligned to 2^K. We found that this benchmark is doing three large allocations. We believe the overhead is mainly due to the different allocation strategies used for the large objects. The memory overhead for this benchmark is -0.04% (Table 5.2, column-2). If we use 2^K aligned sizes, the memory overhead is 0.33% (Table 5.2, column-3).

Overall, these results demonstrate that the custom allocator incurs low CPU overhead for most benchmarks (1.05% geometric mean for CPU SPEC 2017 and 0.21% for Polybench), while maintaining reasonable memory overhead (7.47% and 0.13% geometric means for CPU SPEC 2017 and Polybench, respectively). These results indicate that the allocator overhead is manageable for many large-scale applications.

5.4.2 Performance Benefits without Profiler

We first discuss the performance benefits obtained by applying SCOUT on the programs from Polybench and CPU SPEC 2017 benchmarks in the absence of the profiler. To obtain the performance benefits, we computed the percentage change in the execution time of the benchmark when compiled with (optimized) and without (native) our pass, using the custom memory allocator in both cases.

Polybench benchmarks. For Polybench, we report the performance benefits for two different methods of benchmark compilation. Firstly, we report the performance benefits obtained when compiled with and without our pass along with the custom allocator. Table 5.3 shows the number of versioned loops by SCOUT (column-2 $\#L_a$), and the performance benefits when compiled with and without SCOUT for the six benchmarks showing performance benefits of more than 3% (column-3 P_a). Secondly, we report the performance benefits when compiled with and without the `DPOLYBENCH_USE_RESTRICT` option of Polybench. This option inserts the `restrict` keyword to allow the compiler to assume absence of aliasing for function arguments. Column-4 (P_r) of Table 5.3 shows the performance benefits for the same six benchmarks using the `restrict` keyword. Additionally, this table shows the variance of the execution times for the benchmarks' five runs. Column-5 shows the variance of the execution times for five runs when the benchmarks are compiled with the custom allocator. Column-6 shows the variance of the execution times for five runs when the benchmarks are compiled with the custom allocator and SCOUT. The variance is always less than 0.001.

For Polybench, SCOUT shows similar performance benefits as that of using `restrict` keyword except for two benchmarks, out of 30. We observed that in the case of SCOUT, non-aliasing relationships in the benefited versioned loops involve the function arguments. Therefore, the triggered optimizations in both cases were identical, resulting in similar performance benefits.

We observed a substantial performance degradation when `jacobi-1d` (-1.11%) and `adi` (-5.16%) benchmarks were compiled with the `restrict` keyword. In this case, LLVM failed to preserve non-aliasing information throughout the transformation passes [108]. The loss of non-aliasing information resulted in introducing some extra instructions by loop strength reduction and SLP optimizations, slowing down the performance. Therefore, usage of the `restrict` keyword degraded the performance of these benchmarks. However, SCOUT pre-

served non-aliasing information throughout the transformation passes with the help of custom intrinsic and `no-alias` metadata, resulting in performance benefits of 11.8% and 0.42% for `jacobi-1d` and `adi` benchmarks, respectively. SCOUT enabled more optimization opportunities (i.e., LICM and SLP) for these benchmarks.

```
static void kernel_bicg(int m,
    int n, double A[], double B[],
    double q[], double p[],
    double r[]) {
  ...
  for (i = 0; i < _PB_N; i++) {
    q[i] = SCALAR_VAL(0.0);
    for (j = 0; j < _PB_M; j++) {
      s[j] = s[j] + r[i] * A[i][j];
      q[i] = q[i] + A[i][j] * p[j];
    }
  }
}
```

```
static void kernel_gesummv(
  int n, double alpha,
  double beta, double A[],
  double B[], double tmp[],
  double x[], double y[]) {
  ...
  tmp[i] = SCALAR_VAL(0.0);
  y[i] = SCALAR_VAL(0.0);
  for (j = 0; j < _PB_N;
        j++) {
    tmp[i] = A[i][j] * x[j]
        + tmp[i];
    y[i] = B[i][j] * x[j]
        + y[i];
  }
  y[i] = alpha * tmp[i]
    + beta * y[i];
}
```

```
static void kernel_jacobi_1d(
    int tsteps, int n, double A[],
    double B[]) {
  ...
  for (i = 1; i < _PB_N - 1; i++)
    B[i] = 0.33333 *
        (A[i-1] + A[i] + A[i + 1]);
  for (i = 1; i < _PB_N - 1; i++)
    A[i] = 0.33333 *
        (B[i-1] + B[i] + B[i + 1]);
}
```

```
static void kernel_doitgen(
    int nr, int nq, int np,
    double A[][][],
    double C4[][],
    double sum[]) {
  ...
  sum[p] = SCALAR_VAL(0.0);
  for (s = 0; s < _PB_NP; s++)
    sum[p] +=
        A[r][q][s] * C4[s][p];
  ...
}
```

Figure 5.5: Code snippets from the Polybench benchmarks showing performance benefits of more than 10%.

Out of 30 benchmarks, six benchmarks show performance benefits of more than 3% when compiled with SCOUT, as shown in Table 5.3 (column-3). Out of these six benchmarks, four benchmarks have high performance benefits of more

than 10%. We have listed code snippets from these four benchmarks in Figure 5.5, referred to in the discussion below.

1. In `bicg` benchmark, for the loop in function `kernel_bicg` the compiler could move the read/write access to `q[i]`, and `r[i]` outside the inner loop body.
2. In `gesummv` benchmark, for the loop in function `kernel_gesummv` the compiler could move the read/write access to `tmp[i]`, and `y[i]` outside the loop body.
3. In `jacobi-1d` benchmark, for the loop in function `kernel-_jacobi_1d`, the compiler could move `A[i-1]` and `A[i]` from the first loop outside the loop body and `B[i-1]` and `B[i]` from the second loop outside the loop body. This optimization further allowed the compiler to vectorize both loops more efficiently by reducing the number of read memory accesses from 6 to 2 for a vector width of 4 for each loop.
4. In `doitgen` benchmark, for the loop in function `kernel_doitgen`, the compiler was able to move `sum[p]` outside the loop body.

Out of 30 benchmarks, five benchmarks named, `floyd-warshall`, `nussinov`, `seidal-2d`, `trisolv` and `trmm` did not show any improvement as none of the loops were versioned. These benchmarks consisted of read/write memory accesses with the same base pointers for different array elements. Alves et al. [24] did not report results for the `floyd-warshall` benchmark. However, for the other four benchmarks, Alves et al. [24] did not report any substantial improvements.

For the remaining 19 benchmarks, the performance benefits varied from 0% to 3%. The transformations applied by SCOUT allowed the compiler to trigger optimizations like LICM and GVN. In some cases, these transformations allowed the compiler to generate a better vectorized code. The efficient vectorization improved the overall performance of the benchmarks. Based on the results ob-

Table 5.4: Speedups for Polybench. We report speedups for three benchmarks that resulted in substantial speedups for the hybrid approach [24]. (**S_h = Speedup with the hybrid approach [24], S_{ra} = Speedup when compiled with `restrict` keyword using LLVM-3.6.0, S_s = Speedup with SCOUT, S_{rb} = Speedup when compiled with `restrict` keyword using LLVM-10.**)

Benchmark	S_h	S_{ra}	S_s	S_{rb}
gesummv	2.5	1.92	1.98	1.94
bicg	2.7	2.06	2.05	2.05
gramschmidt	1.4	1.01	1.01	1.01

tained, we can conclude that for Polybench, the performance benefits obtained using the `restrict` keyword and SCOUT are similar in most cases. Although the performance benefits are similar in many cases, using the `restrict` keyword relies on programmer knowledge to guarantee non-aliasing, which is often unavailable or unsafe in large legacy code bases. In contrast, SCOUT automatically infers and enforces disambiguation through runtime checks without requiring source-level annotations, making it applicable to existing programs while preserving correctness.

Table 5.4 shows the results for the three benchmarks that led to a substantial speedup using the hybrid approach [24]. In this table, S_h (column-2) represents the speedups reported for the hybrid approach, S_{ra} (column-3) represents the speedup when we compiled the benchmarks with LLVM-3.6.0 (used in the experiments performed by Alves et al. [24]) using the `restrict` keyword, S_s (column-4) represents the speedups obtained with SCOUT and S_{rb} (column-5) represents the speedups when we compiled the benchmarks with LLVM-10 (used in our experiments) using the `restrict` keyword.

The hybrid (polyhedral and symbolic range analysis) approach [24] resulted in a substantial speedup for the three benchmarks as shown in Table 5.4 (column-2). The speedups using SCOUT (S_s) were lesser than the speedups with the hybrid approach (S_h). To investigate it further, we compiled these benchmarks with the LLVM-3.6.0 version (used by Alves et al. [24]) using the `restrict` keyword

(we used the `restrict` keyword with LLVM-3.6.0 as the artifacts of Alves et al. [24] were not publicly available). The resulted speedups (S_{ra}) were similar to those of SCOUT and with the `restrict` keyword (S_{rb}) using LLVM-10 (used by SCOUT). For all these benchmarks, SCOUT could resolve memory dependencies in loops using dynamic checks. In all cases, SCOUT added runtime checks for the pointer arguments, and thus, we observed a similar set of optimizations using the `restrict` keyword. We expect similar behavior in Alves et al. [24] hybrid approach, with differences in reported speedups likely due to the compiler versions used. Alves et al. used LLVM-3.6.0, whereas we used LLVM-10, whose more advanced optimizations yield better baseline performance and make additional gains from precise alias information appear smaller.

We recommend using the hybrid approach [24] for Polybench benchmarks. For these benchmarks, SCOUT did not find any loop that cannot be versioned using the hybrid approach. The hybrid analysis can disambiguate pointers that involve read/write memory accesses with the same base pointers. SCOUT and the purely dynamic approach [24] fail to disambiguate such memory accesses. Therefore, these techniques fail to benefit the performance in such cases.

Real-world benchmarks such as CPU SPEC 2017 have many loops for which the hybrid approach cannot be applied. Due to the high cost of the purely dynamic approach proposed by Alves et al. [24], these loops cannot be further optimized. Our fast dynamic checks enabled some interesting optimizations in CPU SPEC 2017 that we will discuss next.

CPU SPEC 2017 benchmarks. We now discuss the results for CPU SPEC 2017 benchmarks without using feedback from the profiler. Table 5.5 shows loop-related statistics and performance benefits obtained for CPU SPEC 2017 benchmarks. In this table, $\#L_a$ represents the total number of versioned loops by SCOUT (column-2), $\#L_b$ represents the total number of versioned loops that were not vectorizable before but can be vectorized after the transformations ap-

Table 5.5: Performance benefits for CPU SPEC 2017 without profiler. (#L_a = Total number of versioned loops by SCOUT, #L_b = Total number of versioned loops that are vectorizable, #L_e = Total number of versioned loops executed, P_a = Performance benefits when compiled with SCOUT, V_c = Variance of the execution time when compiled with the custom allocator, V_s = Variance of the execution time when compiled with the custom allocator and SCOUT. Performance-benefit numbers represent percentages.)

Benchmark	#L_a	#L_b	#L_e	P_a	V_c	V_s
namd_r	326	249	9	-0.38	0.7	0.3
parest_r	1020	568	69	-0.3	0.3	0.5
lbm_r	0	0	0	0	0	0
imagick_r	91	24	4	0	0.8	0.3
nab_r	35	15	21	-0.09	0.2	0
perlbench_r	50	18	12	-0.4	1.2	0.7
gcc_r	175	48	35	0.13	1	1.7
mcf_r	0	0	0	0	0	0
omnetpp_r	39	1	6	-1.79	0.2	0.7
xalancbmk_r	286	268	34	0.88	1.3	0.7
x264_r	124	25	32	0	0	0
deepsjeng_r	6	0	6	-0.1	0	0.2
leela_r	1	0	0	0	0	0
xz_r	2	1	0	0	0	0
povray_r	25	7	2	-2.11	0.8	1.6
blender_r	409	89	31	-3.37	0	0.8

plied by SCOUT (column-3), #L_e represents the number of versioned loops executed out of the total versioned loops by SCOUT (column-4), P_a represents the performance benefits obtained without using the profiler (column-5), V_c represents the variance of the execution time for the five runs when the benchmarks are compiled with the custom allocator (column-6), and V_s represents the variance of the execution time for the five runs when the benchmarks are compiled with the custom allocator and SCOUT (column-7).

To obtain the performance benefits, we compiled all the benchmarks with and without our pass using the custom allocator. We then computed the percentage change in the execution time of the benchmarks. Out of 16 benchmarks, six benchmarks named `lbm_r`, `imagick_r`, `mcf_r`, `x264_r`, `leela_r` and `xz_r` reported no improvement in the execution time. For benchmarks `lbm_r` and

`mcf_r`, none of the loops were versioned by SCOUT. For benchmarks `leela_r` and `xz_r`, some loops were versioned, but those loops were never executed.

The percentage change in the CPU time for the rest of the benchmarks lie in the range of -3.37% (`blender_r`) to 0.88% (`xalancbmk_r`), as shown in Table 5.5, column-5. Except for the `xalancbmk_r` and `gcc_r` benchmarks, the other benchmarks incurred negative or no improvement. SCOUT versioned a significant number of loops for these benchmarks; however, the following reasons led to such performance degradation:

1. The code snippet containing the versioned loop never executes (e.g., in the case of `blender_r` only 7% of the versioned loops were executed).
2. The dynamic checks never hold. In such a case, the original version of the loop executes, and the dynamic checks ends up adding an extra overhead (e.g., we found four such loops in case of `gcc_r` and five loops in case of `x264_r`).
3. The overhead of dynamic checks is more than the benefits of the enabled optimizations.

Our observations showed that even though most of the benchmarks did not show a substantial performance improvement, these benchmarks showed the potential to vectorize more loops over the existing implementation. The range of additional vectorized loops lies between 2% to 93% w.r.t. the total versioned loop by SCOUT. Out of 16 benchmarks, for 11 benchmarks, this percentage is more than 10%. For `xalancbmk_r`, `namd_r` and `parset_r` benchmarks 93%, 76% and 55% of the additional versioned loops are vectorizable. These statistics showed that the compiler could vectorize more loops, but the existing approaches failed on such loops.

The purely dynamic approach in Alves et al. [24] slowed down the allocation-heavy `401.bzip2` benchmark from SPEC CPU 2006 by 29%. On the other

hand, the maximum slow down with SCOUT is 10.15% for the `omnetpp_r` benchmark (including allocator overhead) and 3.37% for the `blender_r` benchmark (excluding allocator overhead), as shown in Tables 5.1 (column-5) and 5.5 (column-5). Moreover, it would be unfair to compare these results directly because unlike the purely dynamic approach, SCOUT takes feedback from the static optimizations to reduce the number of dynamic checks and decide which loops can be benefited from versioning. Alves et al. [24] did not report results for other SPEC CPU 2006 benchmarks (apart from `401.bzip2`).

Overall, the results without profiling show that SCOUT delivers substantial performance improvements for a subset of benchmarks, while also revealing the limits of loop versioning. For Polybench, performance gains are observed for a small number of benchmarks with safely versioned loops. However, since most Polybench loops are effectively handled by hybrid disambiguation techniques that require no custom allocator, those approaches are generally better suited. For CPU SPEC 2017, although the speedups are often limited, SCOUT versions many loops that existing techniques cannot handle, exposing additional vectorization opportunities. These results motivate the selective use of runtime disambiguation via profiling, discussed in the next subsection.

5.4.3 Performance Benefits with Profiler

We estimated performance benefits of the CPU SPEC 2017 benchmarks based on the profiler's feedback. To obtain the performance benefits with the profiler, we computed the percentage change in the execution time of the benchmark when compiled with only those versioned loops that showed some improvement in the execution time w.r.t. the native compilation. Firstly, the benchmarks were compiled and executed with the optimized (i.e., versioned loops) and then with the unoptimized versions of the loops to obtain the profile files. SCOUT identified loops that showed some improvement using the generated profile files. The different thresholds for performance improvement determined the benefited loops.

Table 5.6: Performance benefits for CPU SPEC 2017 with profiler. ($\#L_p$ = Total number of versioned loops based on the profiler's feedback, $\#L_b$ = Total number of versioned loops that are vectorizable, P_a = Performance benefits when compiled with SCOUT, τ = Threshold for improvement in the execution time of the loop, S_{rec} = SCOUT recommended (Y = Yes, N = No). Performance-benefit numbers represent percentages.)

	τ = 0			τ = 10			τ = 20			τ = 30			
Benchmark	$\#L_p$	$\#L_b$	P_a	$\#L_p$	$\#L_b$	P_a	$\#L_p$	$\#L_b$	P_a	$\#L_p$	$\#L_b$	P_a	S_{rec}
namd_r	9	1	0.51	5	1	0.13	0	0	0	0	0	0	Y
parest_r	24	17	0.31	18	13	0.73	15	12	0.43	12	10	0.31	N
lbm_r	0	0	0	0	0	0	0	0	0	0	0	0	N
imagick_r	1	0	0.23	1	0	0.23	1	0	0.23	1	0	0.23	N
nab_r	12	9	0.27	11	9	0	11	9	0	8	7	0	N
perlbench_r	0	0	0	0	0	0	0	0	0	0	0	0	N
gcc_r	5	4	0.62	2	2	0.73	2	2	0.73	0	0	0	Y
mcf_r	0	0	0	0	0	0	0	0	0	0	0	0	N
omnetpp_r	3	0	0.5	1	0	0.25	0	0	0	0	0	0	N
xalancbmk_r	10	9	1.32	4	4	1.47	2	2	0.9	2	2	0.9	N
x264_r	19	1	0.89	12	1	0.45	9	1	0.34	3	1	0.56	Y
deepsjeng_r	2	0	0.49	0	0	0	0	0	0	0	0	0	N
leela_r	0	0	0	0	0	0	0	0	0	0	0	0	N
xz_r	0	0	0	0	0	0	0	0	0	0	0	0	N
povray_r	0	0	0	0	0	0	0	0	0	0	0	0	N
blender_r	11	0	0.52	3	0	0.31	0	0	0	0	0	0	Y

Table 5.6 shows the loop-related statistics and performance benefits for thresholds (τ) 0%, 10%, 20% and 30%. In this table, $\#L_p$ represents the total number of versioned loops based on feedback from the profiler, $\#L_b$ represents the number of versioned loops not vectorizable before but vectorized after the transformations applied by SCOUT, P_a represents the performance benefits obtained using feedback from the profiler and S_{rec} represents whether we recommend using SCOUT for the corresponding benchmark (Y represents we recommend using SCOUT and N represents we do not recommend using SCOUT). We recommend using SCOUT for those benchmarks where the CPU time overhead of the custom allocator (Table 5.1, column-5) is lesser than the performance benefits obtained with the profiler for at least one of the thresholds. Table 5.7 shows the variance of the execution time for five runs of the benchmarks for different thresholds. V_c represents the variance when compiled with the custom allocator, V_s represents the variance when compiled with the custom allocator, and SCOUT.

Table 5.7: Variance of the execution time for five runs of the CPU SPEC 2017 benchmarks. τ = Threshold for improvement in the execution time of the loop, V_c = Variance of the execution time when compiled with the custom allocator, V_s = Variance of the execution time when compiled with the custom allocator and SCOUT.)

		$\tau = 0$	$\tau = 10$	$\tau = 20$	$\tau = 30$
Benchmark	V_c	V_s	V_s	V_s	V_s
namd_r	0.7	0	0.8	0.7	0.7
parest_r	0.3	0.5	0.8	0.8	0.3
lbm_r	0	0	0	0	0
imagick_r	0.8	0.7	0.7	0.7	0.7
nab_r	0.2	0.2	0.7	0.7	0.7
perlbench_r	1.2	1.2	1.2	1.2	1.2
gcc_r	1	1	0.2	0.2	1
mcf_r	0	0	0	0	0
omnetpp_r	0.2	0.7	0.2	0.2	0.2
xalancbmk_r	1.3	1.3	0.3	0.7	0.7
x264_r	0	0.3	0.2	0.3	0
deepsjeng_r	0	0	0	0	0
leela_r	0	0	0	0	0
xz_r	0	0	0	0	0
povray_r	0.8	0.8	0.8	0.8	0.8
blender_r	0	0	0.3	0	0

For the benchmark `xalancbmk_r`, out of 286 versioned loops by SCOUT, only ten loops resulted in some performance improvement based on the statistics shown in Table 5.6 for threshold 0%. After discarding non-benefited loops, the performance benefits increased to 1.32% from 0.88% for threshold 0%. The performance further increased to 1.47% when the four loops with an improvement of more than 10% were versioned. However, the performance drops to 0.9% for the 20%, and 30% threshold as SCOUT only versioned two loops based on the feedback. Out of the ten loops that benefited, the compiler vectorized nine loops based on the transformations applied by SCOUT. This number was further reduced to four for threshold 10% and two for thresholds 20% and 30%, affecting the benchmark's performance. Even though the `xalancbmk_r` benchmark shows the maximum performance benefits, we do not recommend using SCOUT for this benchmark as the allocator's overhead for this benchmark is higher (3.05%, Table 5.1, column-5).

For benchmark `x264_r`, out of 124 versioned loops by SCOUT, only 19 loops benefited based on the statistics obtained by the profiler. This benchmark led to a performance improvement of 0.89% when non-benefited loops were not versioned. For this benchmark, the performance benefits vary with the threshold increase. Moreover, the versioning of non-benefited loops due to the absence of the profiler hampered the performance of this benchmark. The obtained results showed that this benchmark could lead to a performance improvement of around 0.89% using the feedback. For this benchmark, the allocator is also showing an improvement of 2.36% (Table 5.1, column-5). Therefore, we recommend using SCOUT for this benchmark.

The performance benefits for the `gcc_r` benchmark increased from 0.13% to 0.62% when the benchmark was compiled with versioning only those loops that showed some improvement. There were only five such loops out of 175 loops versioned by SCOUT. The performance benefits increased to 0.73% for thresholds 10% and 20%. For the threshold 30%, none of the loops got benefited. We recommend using SCOUT for this benchmark as the maximum performance benefit is 0.73%, and the allocator shows an improvement of 0.98% (Table 5.1, column-5).

For some benchmarks, the performance benefits decreased with the increase in threshold, such as `namd_r`, `omnetpp_r` and `deepsjeng_r` as the number of versioned loops reduces to 0. However, for benchmarks such as `parest_r`, `gcc_r` and `xalancbmk_r`, the performance improved for threshold 10% compared to threshold 0%. The possible reason behind this could be that the instrumentation of *rdtsc* instruction for collecting timing statistics during the profile phase changed the behavior of some loops. The threshold 0% setup also includes loops that show minor improvements over the unoptimized version during the profile phase. However, the benchmarks showed higher performance benefits when such loops were not versioned for the threshold 10%.

We observed the benchmarks were benefited from the profiler because it helped filter out some non-benefited loops. The user can choose the threshold according to their requirements to obtain maximum performance benefits using SCOUT. We recommend using SCOUT for four of the CPU SPEC 2017 benchmarks named, `namd_r`, `gcc_r`, `x264_r`, and `blender_r`. We conclude that with the help of the profiler, SCOUT can identify the loops more effectively that were ultimately benefited and version only those loops to get maximum performance benefits.

5.4.4 Code Patterns Optimized using SCOUT for CPU SPEC 2017

Figure 5.6 shows some of code snippets from CPU SPEC 2017 benchmarks for which loop bounds are not loop invariants. These loops were executed during runtime and yielded more than 20% improvement. However, SCOUT could optimize many such loops during compile time. In most cases, we found that structure fields or class elements are accessed in the loop condition, involving memory access. This looks like a common coding pattern that exists in real-world workloads. The compiler could not compute the static bounds of these loops due to unresolved memory dependencies. Note that none of the existing techniques can be applied to this loop except the purely dynamic approach proposed in Alves et al. [24] that would require $O(log\ n)$ checks.

5.5 Summary

This chapter introduced a novel memory allocator design that enables constant-time pointer disambiguation at runtime using a single memory access. By constraining the size and alignment of memory objects during allocation, our approach performs lightweight dynamic checks. Prior work by Alves et al. [24] proposes a purely dynamic disambiguation approach based on object tags, but requires red-black tree lookups, leading to logarithmic overheads ($O(logn)$) that limit scalability. Our work is motivated by this approach but reduces the runtime overhead of dynamic checks through constant-time disambiguation. Unlike LLVM's loop-versioning mechanism based on scalar evolution analysis, which

```
//544.nab_r
//mme34()
//line 1978
for (i = -1;
    i < prm->Natom;
    i++) {
  iexw[eoff + i]
    = -1;}
//Improvement: 60%
```

```
//544.nab_r
//nbond()
//line 866
for (i = -1;
    i < prm->Natom;
    i++) {
  iexw[i] = -1;}
//Improvement: 43%
```

```
//544.nab_r
//set_belly_mask()
//line 1980
for( i=0;
    i<prm->Natom;
    i++ )
  prm->N14pairs[i]
    = 0;
//Improvement: 38%
```

```
//544.nab_r
//readparm()
//line 1494
for (i = 0;
    i < prm->Natom;
    i++)
  prm->N14pairs[i]
    = 0;
//Improvement: 37%
```

```
//544.nab_r
//mme_init()
//line 1292
for (i = 0;
    i < prm->Natom;
    i++) {
  pairlist[i] = NULL;
  lpairs[i]
    = upairs[i] = 0;}
//Improvement: 33%
```

```
//544.nab_r
//mme_init()
//line 1224
for (i = 0;
    i < prm->Natom;
    i++) {
  pairlist2np[i]
    = NULL;
  lpairs2np[i]
    = upairs2np[i]
    = 0;}
//Improvement: 32%
```

```
//523.xalancbmk_r
//expand()
//line 620
//fElemCount is a
//class member.
for (unsigned int
    index = 0; index
    < fElemCount;
    index++)
  newList[index]
  = fRanges[index];
//Improvement: 51%
```

```
//525.x264_r
//FmoGenerateMb-
//ToSliceGroupMap()
//line 149
for (i=0; i<p_Vid->
  PicSizeInMbs; i++) {
  MbToSliceGroupMap++
    = *MapUnitTo-
    SliceGroupMap++;
}
//Improvement: 52%
```

```
//510.parest_r
//get_dof_indices()
//line 2043
for (unsigned int
    i=0;
    i< accessor
    .get_fe()
    .dofs_per_cell;
    ++i, ++cache)
  dof_indices[i]
    = *cache;
//Improvement: 77%
```

```
//502.gcc_r
//df_worklist-
//_dataflow()
//line 1023
for (i = 0;
    i < cfun->cfg->
    x_last_basic_block;
    i++)
  bbindex_to
  _postorder[i]
  = cfun->cfg->
  x_last_basic_block;
//Improvement: 29%
```

```
//502.gcc_r
//init_graph()
//line 1116
for (j = 0;
    j < graph->size;
    j++) {
graph->rep[j] = j;
  graph->pe_rep[j]
    = -1;
  graph->
    indirect_cycles[j]
    = -1;}
//Improvement: 22%
```

```
//538.imagick_r
//ParseGeometry()
//line 967
while (isspace((int)
  ((unsigned char)
  *p)) != 0)
  p++;
//Improvement: 51%
```

Figure 5.6: Code snippets from the CPU SPEC 2017 benchmarks that show substantial improvement.

only works for loops with invariant bounds and affine memory accesses, our technique supports a broader class of loops, including those with non-invariant bounds that LLVM cannot handle. This expands the set of loops that can be optimized, enabling not just vectorization but also other transformations like loop-invariant code motion, dead store elimination, and load elimination, which LLVM does not typically attempt under imprecise aliasing conditions. Our technique uses a feedback-guided mechanism to identify only the dynamic checks necessary to enable optimizations. Our approach scaled to five real-world applications (CPU SPEC 2017). The CPU and memory overheads of this scheme for CPU SPEC 2017 benchmarks lie in the range -2.57% to 8.36% and -2.29% to 34.89%. These results demonstrate the practicality and performance benefits of our method in real-world settings. However, the allocator's size and alignment constraints introduce overhead that limits scalability for some workloads. A key direction for future work is to reduce these allocator overheads while preserving the efficiency of constant-time pointer disambiguation.

Chapter 6

A Region-Based Allocator

Our previous work (Scout Chapter 5) presented a scalable solution to the limitations of static alias analysis by enabling constant-time pointer disambiguation through segment-based memory allocation. The proposed tool, Scout, disambiguates two pointers `x` and `y` by computing the size of the object pointed to by either of them, say `len`. If `x` and `y` are at least `len` apart, they cannot overlap. Scout retrieves the object size from a given pointer in just one memory access, enabling highly efficient runtime checks. Compared to prior techniques such as Alves et al. [24], the dynamic checks in Scout are significantly faster and scale to large applications, demonstrating measurable performance improvements in a few CPU SPEC 2017 benchmarks.

However, a major drawback of Scout is the additional CPU and memory overhead introduced by the allocator due to increased object size. Scout enforces that all allocations be rounded up to the nearest power of two (2^k), regardless of whether they are involved in versioned loops. As a result, the CPU and memory overheads for CPU SPEC 2017 benchmarks range from –2.57% to 8.36% and –2.29% to 34.89%, respectively (with negative values indicating performance improvements). These overheads stem from the fact that Scout constrains all allocations, even those never accessed within versioned loops.

In this chapter, we present our tool, RAPID. RAPID further reduces the over-

head of allocator and dynamic checks by only constraining those memory allocations that may be accessed in loops that undergo additional versioning in the approaches proposed by Alves et al. [24] and Scout (Chapter 5). At the high level, RAPID ensures that the conflicting memory accesses always belong to different heap regions, and the loop-versioning condition simply checks if the pointer addresses belong to different regions. The heap regions are allocated in a way that the top 32-bits in the virtual addresses corresponding to two different regions are always different. Thus, the dynamic check for disambiguation is very efficient and doesn't require any memory access.

RAPID works in two phases. In the first phase, RAPID uses a profiler to correlate conflicting memory accesses in the loops to their allocation sites. In the second phase, RAPID generates code in a way that conflicting memory accesses are allocated from different heap regions (at runtime) and implement loop-versioning.

Here are the key differences between RAPID and Scout. Scout requires one memory access to disambiguate two pointers, whereas RAPID doesn't require any memory access in the dynamic checks for disambiguation. Scout constrains the size and alignment of all objects, which is the primary source of its overhead. RAPID doesn't change the allocation size and alignment of objects. Also, the different regions are assigned to only those objects that may be accessed in the versioned loop – not to all objects. Therefore, the allocator overhead in RAPID is very low compared to Scout. The allocator's CPU and memory overheads of RAPID for CPU SPEC 2017 benchmarks lie in the range -3.67% to 0.46% and -0.62% to 6.96%, respectively. Due to the low overhead of allocator and dynamic checks, RAPID could improve the performance of 12 out of 16 CPU SPEC 2017 benchmarks. In four benchmarks, none of the loops were versioned using the dynamic check.

This work makes the following contributions:

1. It introduces an approach that combines loop-versioning with lightweight dynamic checks. Our approach includes a region-based memory allocator and instruments region-based dynamic checks to decide the version of the loop to execute at runtime.
2. A profiler to identify conflicting memory accesses inside the loops and potential loop candidates that show potential for improvement.
3. Integration of the approach with LLVM.
4. Evaluation of the performance benefits of Polybench and CPU SPEC 2017 benchmarks using the approach.

6.1 Motivating Example

We now discuss the overview of our approach and compare it with existing techniques using the example in Figure 6.1. This example is similar to the one we used in our Scout (Chapter 5), and to the examples presented by Alves et al. [24]. We found similar code patterns in CPU SPEC 2017 benchmarks as discussed in Section 6.4.

The example shown in Figure 6.1 adds the elements of arrays `b` and `c` and stores the result in array `a` (at line 4-6). The compiler can vectorize this loop if it knows for sure that `b`, `c`, and `size` don't overlap with `a`. Line 63 shows the vectorized loop body. Here, ``i:i+3'' means four operations at indices `i, i+1, i+2, i+3` are performed in parallel instead of one operation at line 5. Notice that if array `a` overlaps with either array `b`, array `c`, or `size`, then the parallel execution may yield a different result than the sequential execution. In this case, the alias relationships between these objects depend on the values passed by the caller and hence can't be answered using the intraprocedural alias analysis. As discussed before, the polyhedral, symbolic range, and scalar evolution analyses require the loop bounds to be loop invariants, which is not true in this case. The loop bound `*size` is not a loop invariant as `size` can overlap with `a`, which might end up modifying the loop bounds inside the loop body. Therefore, in addition to RAPID, the pointer disambiguation technique discussed

in Alves et al. [24], and Scout can be used to version this loop.

```
1. void foo(int *a, int *b,
2.          int *c, int *size)
3. {
4.   for (int i = 0; i < *size;
       i++) {
5.    a[i] = b[i] + c[i];
6.   }
7. }
```

(a) Original `foo`.

```
8. int main() {
9.   int *a = (int*)malloc(36);
10.  int *b = (int*)malloc(36);
11.  int *c = (int*)malloc(36);
12.  int size = 9;
13.  foo(a, b, c, &size);
14.  ...
15.}
```

(b) Original `main`.

Figure 6.1: An example to discuss RAPID approach.

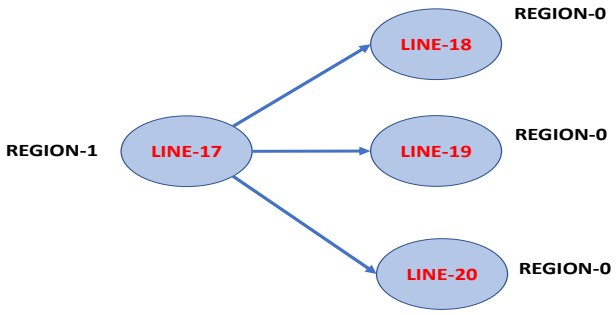


Figure 6.2: Interference graph used for assigning regions. Different regions are assigned to nodes connected using an edge.

Figure 6.3d shows the versioned loop using RAPID. Lines 58-60 correspond to the loop-versioning condition. RAPID ensures that conflicting memory accesses `(a, b)`, `(a, c)`, and `(a, size)` belong to different heap regions. Two pointer addresses belong to different heap regions if the top 32-bits in the addresses are different. The condition at line 58 is true at runtime if `a` and `b` belong to different regions. Similarly, conditions at lines 59 and 60 hold at runtime if `(a, c)` and `(a, size)` are in different regions. Notice that `b`, `c`, and `size` can be in the same region. If the loop condition fails, the original loop at lines 69-71 executes.

RAPID works in two phases. In the first phase, it profiles the application to find out the allocation sites of pointer variables that potentially require a disambiguation check. In this example, dynamic checks are required for pointer pairs `(a, b)`, `(a, c)`, and `(a, size)`. Figure 6.3a shows the `main` routine instrumented for profiling. At lines 17-19, `malloc` is replaced with `malloc_prof`

```
16. int main() {
17.  int *a = (int*)malloc_prof
       (36, 17);
18.  int *b = (int*)malloc_prof
       (36, 18);
19.  int *c = (int*)malloc_prof
       (36, 19);
20.  int *size_addr = (int*)
       malloc_prof(4, 20);
21.  size_addr[0] = 9;
22.  foo(a, b, c, size_addr);
23.  ...
24. }
```

(a) Profiling phase of RAPID for `main`.

```
25. int main() {
26.  int *a = (int*)rmalloc
       (36, 1);
27.  int *b = (int*)malloc(36);
28.  int *c = (int*)malloc(36);
29.  int size = 9;
30.  foo(a, b, c, &size);
31.  ...
32. }
```

(b) Final code generated for `main` using RAPID.

```
33. void foo(int *a, int *b,
34.   int *c, int *size)
35. {
36.  if (start_addr(a)
       != start_addr(b) &&
37.     start_addr(a)
       != start_addr(c) &&
38.     start_addr(a)
       != start_addr(size)) {
39.   log_pair(a, b);
      log_pair(a, c);
40.   log_pair(a, size);
41.   int t = *size, i;
42.   for (i = 0; i + 3 < t;
        i += 4) {
43.     a[i:i+3] = b[i:i+3]
          + c[i:i+3];
44.   }
45.   for (; i < t; i++) {
46.     a[i] = b[i] + c[i];
47.   }
48.  } else {
49.   for (int i = 0;
        i < *size; i++) {
50.    a[i] = b[i] + c[i];
51.   }
52.  }
53. }
```

(c) Profiling phase of RAPID for `foo`.

```
54. #define D (1ULL << 32)
      /* 4GB */
55. void foo(int *a, int *b,
56.   int *c, int *size)
57. {
58.  if ((a ^ b) >= D &&
59.      (a ^ c) >= D &&
60.      (a ^ size) >= D) {
61.   int t = *size, i;
62.   for (i = 0; i + 3 < t;
        i += 4) {
63.    a[i:i+3] =
        b[i:i+3] + c[i:i+3];
64.   }
65.   for (; i < t; i++) {
66.    a[i] = b[i] + c[i];
67.   }
68.  } else {
69.   for (int i = 0;
        i < *size; i++) {
70.    a[i] = b[i] + c[i];
71.   }
72.  }
73. }
```

(d) Optimized `foo` using RAPID.

Figure 6.3: Transformed code with RAPID.

that additionally takes the allocation site (a unique identifier for each allocation site, e.g., line number) and stores the allocation site in the metadata corresponding to the allocated object. Lines 36-38 show the loop condition of the versioned loop during the profile phase. It uses the `start_addr` API, which returns the starting address of an object. Two objects can not overlap if the starting addresses are different. If the optimized path is taken at runtime, the allocation sites of the conflicting pointer pairs are logged. For example, (17, 18), (17, 19), and (17, 20) are logged at lines 39 and 40 (notice that the allocation sites of `a`, `b`, `c`, and `size` are 17, 18, 19, and 20). The next goal is assigning different regions to each conflicting pair of allocation sites. This is done using an interference graph. In an interference graph, there is an edge between two allocation sites if a disambiguation check is required for objects allocated at these sites. Figure 6.2 shows the interference graph for this example. RAPID assigns regions to each of the nodes in a way that nodes connected using an edge don't have the same region. We have used the standard graph coloring algorithm to assign regions to nodes in the interference graph. In this example, the graph coloring algorithm assigns region-id zero to allocation sites 18, 19, and 20 and region-id one to allocation site 17.

In the second phase, `malloc` is replaced with `rmalloc` if the region-id of the corresponding allocation site is not zero. `rmalloc` additionally takes the target region-id. The default region-id of all allocations is zero. Because, in this case, the region-id for array `a` is one, RAPID replaces the `malloc` with `rmalloc` at line 26. The rest of the allocations are in region-id zero and are not replaced. The optimized code for `foo` is shown in Figure 6.3d that implements faster checks in the loop-versioning condition.

In Alves et al. [24] approach, the allocation sites are not changed. The dynamic checks at lines 58-60 require expensive red-black tree lookups to compute the starting address of `a`, `b`, `c`, and `size`. Scout sets the allocation size and alignment of objects at lines 26-28 to 48 (a power-of-two). The dynamic checks at

lines 58-60 require memory access to obtain the size of object `a`.

RAPID enables the compiler to explore useful optimization opportunities missed by the compiler, such as code vectorization, loop invariant code motion, load elimination, and store elimination. In the next section, we describe our approach in detail.

6.2 RAPID

This section describes the RAPID approach in detail. Figure 6.4 shows the architecture of RAPID. RAPID uses region-based allocator that enables faster disambiguation checks. RAPID compiles the application in two phases. RAPID takes an intermediate representation (IR) of the program as input in both phases. The first phase is the profile phase. In the profile phase, RAPID identifies and instruments loops that can be further optimized using loop-versioning. The IR is instrumented to log the allocation sites of the conflicting memory accesses inside the loop. The IR is then transformed into an executable that is executed on a given set of inputs to generate a log file. The second phase also takes the log file generated in the first phase as input. In the second phase, RAPID implements the graph coloring algorithm to infer region-ids for conflicting allocation sites, modifies allocation sites to allocate from inferred region-ids, and implements loop-versioning. The final code generated by RAPID is linked to the region-based allocator during the execution. Now, we discuss both phases in detail.

6.2.1 Region-based allocator

Figure 6.5 shows a simplified view of the heap layout in region-based allocator. A region contains a list of segments. A segment is a 4GB (configurable at compile time) contiguous memory area. The starting address of a segment is always aligned to 4GB. Segments in the regions are disjoint. The segments corresponding to a region can be scattered across virtual address space (for simplicity, we showed them contiguous). Two pointer addresses point to different

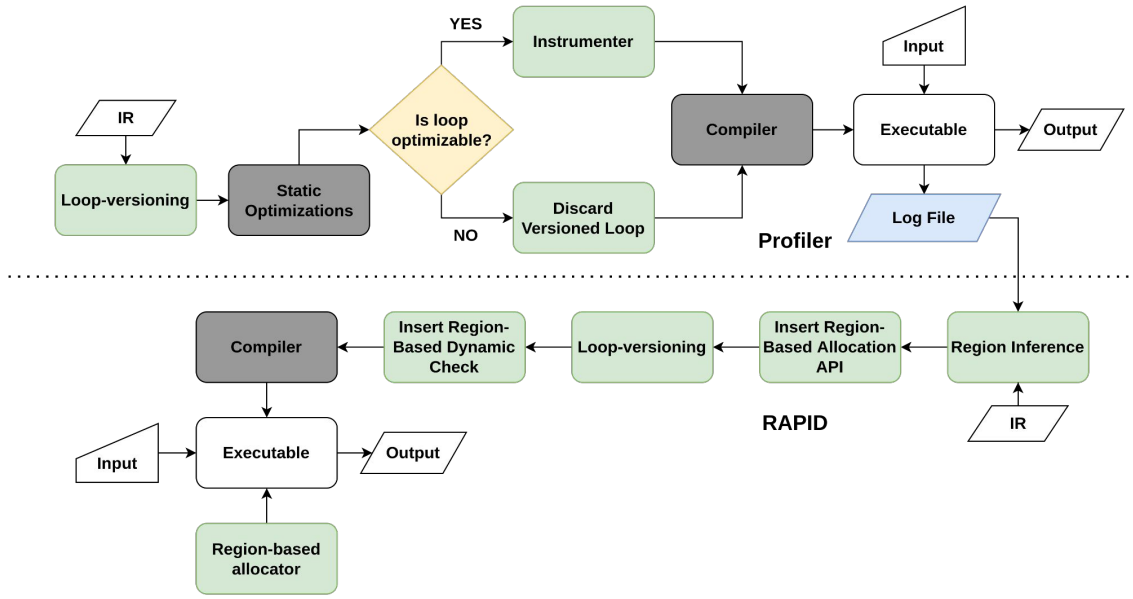


Figure 6.4: Architecture of RAPID.

objects if they belong to different segments or regions. Objects in different regions always belong to different segments. Because of the alignment property of segments, if two objects belong to different segments, then the top 32-bits in their virtual addresses can't be the same. This property enables faster checks to disambiguate pointers belonging to different regions, as we use in our dynamic checks.

6.2.2 Profiling phase

We use the profiler for the following goals:

1. To capture allocation sites of objects accessed inside the loops.
2. To identify potential loop candidates that can be further optimized.
3. To compute the benefit of loop-versioning by counting the number of iterations of the optimized and unoptimized version of the loop.

Identifying allocation sites. To capture allocation sites of objects accessed inside the loop, RAPID inserts an eight-byte object header before the starting address of the object. RAPID assigns a unique integer id to each allocation site,

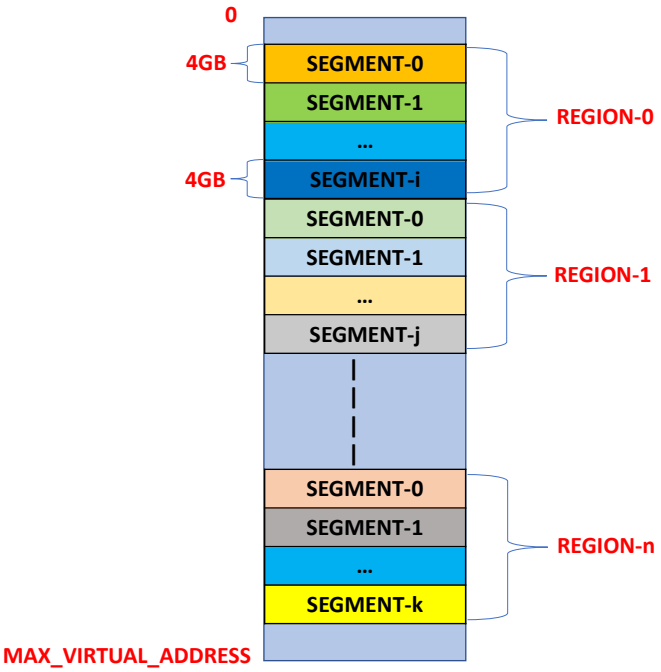


Figure 6.5: Heap layout in region-based allocator.

called allocation-id, which is stored in the object header at the time of allocation. From a given pointer address, the allocation site is obtained by first computing the starting address of the object and then reading the allocation id from the object header.

Loop-versioning. To identify potential candidates for loop-versioning, RAPID follows an approach similar to Scout (Chapter 5). The loop-versioning algorithm is shown in Algorithm 2. `InstrumentLoop` returns true if the loop can be optimized using dynamic checks. `ComputeDataDependencies` collects all the pointer operands that are accessed in a loop. The read-write and write-write pairs for which the static alias analysis fails are added to the `ConflictSet`. It also tries to check if the pointer operands are derived from the same base, in which case they are not added to the `ConflictSet`. Thus, `ComputeDataDependencies` returns a set of pointer pairs (`ConflictSet`) that are accessed inside the loop and follows the following properties:

- Pointer pairs are not the result of pointer arithmetic on the same array in

```
Function InstrumentLoop(Function F, Loop L):
    ConflictSet ← ComputeDataDependencies(L);
    /* ConflictSet contains conflicting pointer pairs */
    if ConflictSet.empty() then
        return false;
    end
    L_IsVectorizable ← CanVectorize(L);
    LV ← CreateCopy(L);
    foreach P ∈ ConflictSet do
        assume P doesn't conflict in LV;
    end
    /* check if we can vectorize the versioned loop
    * assuming pointer pairs don't overlap.
    */
    LV_IsVectorizable ← CanVectorize(LV);
    LICM(LV);
    GVN(F);
    DSE(F);
    if LV and L are the same then
        /* LICM, GVN, DSE didn't work */
        if not LV_isVectorizable or L_IsVectorizable then
            /* Loop L can't be optimized further */
            DeleteLoop(LV);
            return false;
        end
    end
    if not LV_IsVectorizable then
        ConflictSet ← RemoveRedundantChecks(ConflictSet, L, LV);
    end
    /* LV is more optimized than L */
    InsertLoopVersioningCondition(L, LV, ConflictSet);
    InsertCodeToLogAllocationIds(LV, ConflictSet);
    InstrumentLoopIterCount(L, LV);
    return true;
```

Algorithm 2: The profiler takes a function and the loop. It returns whether the loop has a potential to be optimized further. The profiler instruments the conflicting memory accesses present in the functions to store the allocation-ids. It also instruments the preheader of the selected loops based on the feedback from static optimizations to log the required information.

the current function scope.

- At least one of them is involved in a write operation inside the loop body.

- Intra-procedural alias analysis doesn't know precisely if the pointer pairs are alias or not.

`CanVectorize` uses alias analysis and scalar evolution [77, 78] to return true if a loop can be vectorized (no read-write or write-write conflict) with/without loop-versioning. We used the existing implementation in LLVM for `CanVectorize`. Thus, `CanVectorize` routine returns true if the compiler can vectorize the loop. Notice that `InstrumentLoop` takes an optimized IR as input that is not vectorized. The reason is that once the loop is vectorized, the LLVM compiler can't optimize it further. At line 8, a copy of loop, `LV`, is created. At lines 9-11, RAPID assumes that the pointer pairs returned by `ComputeDataDependencies` don't overlap in `LV`. Non-overlapping memory accesses inside the loop may enable other optimizations, such as loop invariant code motion (LICM), global value numbering (GVN), dead store elimination (DSE), and loop vectorization. RAPID performs LICM (line 16), GVN (line 17), and DSE (line 18) on `LV` to check if any of them are enabled. If yes, the loop is versioned. Otherwise, if the original loop (`L`) was not vectorizable and `LV` is vectorizable (due to non-overlapping memory accesses), then also the loop is versioned. At line 31, the loop-versioning condition is inserted. For every pointer pair `(x, y)` in the `ConflictSet`, `InsertLoopVersioningCondition` instrument code to obtain the starting addresses of objects pointed by `(x, y)`. The optimized loop version (`LV`) is executed at runtime if the starting addresses don't match for all pointer pairs in `ConflictSet`. In `LV`, RAPID instruments code to log the allocation-id pairs corresponding to all pointer pairs in the `ConflictSet`. The allocation-id is obtained from the object header of the corresponding object.

Finally, RAPID instruments code to collect the loop-taken statistics for the versioned loop. The loop is versioned in the second phase if the optimized path is taken frequently during profiling. Notice that if `LV` is not vectorizable, we may not need all dynamic checks. For example, at line 61 in Figure 6.1, `size` is

moved outside the loop. If the rest of the loop is not vectorizable, we don't need to check if pointer pairs `(a, b)` and `(a, c)` overlap. This optimization is sound as long as `size` doesn't overlap with `a`. To compute the minimum number of checks, RAPID adopted the approach followed by Scout (Chapter 5). At line 28, `RemoveRedundantChecks` removes the redundant checks using the following approach:

- Initially, mark all pointer pairs in the `ConflictSet` as invalid.
- If a memory access operation (read/write) of pointer p in `L` is not present in `LV`, mark all pointer pairs that contain p in `ConflictSet` as valid.
- Remove all pointers pairs from the `ConflictSet` that are invalid.

Finally, the instrumented code for profiling is converted into an executable. The profiler is executed on user-supplied input to generate a log file that is used in the second phase of RAPID.

6.2.3 Final code generation

In the second phase, RAPID does the following tasks.

- Computes region-id for each allocation site.
- Implements loop-versioning for loops identified in the profiling phase.

Region inference. The goal of the region inference is to compute the region-ids for all allocation sites. The log file generated during the profile phase contains all allocation site pairs that should not belong to the same region. As discussed before, RAPID generates an interference graph in which the nodes are the allocation sites, and an edge between two nodes represent that the nodes should not belong to the same region. Now, the problem of computing region-ids is the same as the graph coloring problem used for register allocation [86]. We used the following algorithm to compute the region-ids, where `n` is the number of nodes in the interference graph.

```
for (i = 2; i <= n; i++) {
  if (graph is colorable using i colors) {
     assign colors starting from zero
     return;
  }
}
```

In this algorithm, colors represent regions. After the region-ids are assigned to each node in the interference graph, all the allocation sites that are present in the interference graph and have region-id other than zero are modified to allocate from the corresponding region.

6.2.3.1 Handling conflicting allocation sites

The allocation sites of two conflicting memory accesses in a versioned loop may be the same if the corresponding allocations are inside a loop. We explain this using the following example. If we replace `a`, `b`, `c` in Figure 6.1b with `arr[0]`, `arr[1]`, and `arr[2]` and allocate them inside a loop, there will be only one `malloc` statement. Now, we can't replace `arr[0]` in Figure 6.3b to allocate from region-1 because the same statement is used to allocate `arr[1]` and `arr[2]`.

In this case, the dynamic check will always fail if we assign a unique region-id to the corresponding allocation site inside the loop. To handle this case, instead of assigning a single region-id, RAPID assigns a range of region-ids to the allocation site. For these cases, during memory allocation, the allocation-id is chosen from the set of assigned region-ids in a round-robin manner. Notice that this approach doesn't guarantee that the dynamic checks for disambiguation always succeed; however, the probability of failure decreases as we increase the set of ids assigned to these allocation sites. Because a large number of regions can degrade an application's performance, we ask the user to provide the maximum number of regions using a compile-time option. After assigning

region-ids to non-conflicting allocation sites, we divide the remaining regions equally among all conflicting allocation sites. In our experiments, we found that a small number of ids are sufficient for the existing cases in real-world benchmarks.

6.2.3.2 Loop-versioning

The loop-versioning algorithm is similar to the algorithm used by the profiler (Algorithm 2), except for a few changes. The dynamic checks in the loop-versioning condition are very efficient. It checks if the pointer pair belongs to different segments by comparing the top 32-bits. The starting address of a segment is always aligned to 4GB. If the `xor` of two pointer addresses is greater than or equal to 4GB (maximum size of a segment), then they belong to different segments and thus different objects. RAPID doesn't log anything in this phase. The loop is versioned only if the optimized path is taken frequently during the profile run.

6.3 Implementation

We implemented RAPID as a part of LLVM infrastructure (v10.0.0). RAPID instruments an already optimized IR that is not vectorized. This is because LLVM can't further optimize an already vectorized loop. We have modified Mimalloc (v 2.0.6) [109, 110] to implement our region-based memory allocator. The primary motivation for picking Mimalloc is that it already supports thread-local heaps that can be easily extended to multiple heap regions.

Mimalloc. `Mimalloc` [109, 110] is a high-performance memory allocator designed for speed, scalability, and low fragmentation, especially in multi-threaded environments. The performance of Mimalloc is comparable to the state-of-the-art allocators. Its architecture is built around three key components: `heaps`, `segments`, and `pages`. Each thread is assigned its own `heap`, reducing contention by allowing most memory operations to occur without synchronization. A heap consists of one or more `segments`, which are large contiguous memory blocks acquired from the operating system. These segments are further di-

vided into `pages`, each responsible for managing allocations of a specific size class.

During allocation, the thread first tries to serve the request from its local page's free list. If the page is full, a new one is fetched from an available segment or a new segment is allocated via `mmap`. For deallocation, if the memory is freed by the same thread that performed the allocation (i.e., within the same heap), it is directly added back to the page's local free list. If a different thread performs the deallocation, the freed memory is placed into the page's remote free list, which is processed later in batches to minimize synchronization overhead. This layered and thread-aware design allows `Mimalloc` to maintain high performance and efficient memory usage across diverse workloads.

We extended Mimalloc to implement thread-local heaps for multiple regions. By default, there is only one region; in that case, Mimalloc uses just one thread-local heap. Mimalloc uses `mmap` to allocate mi-segments. In our implementation, mi-segments are allocated from segments (recall that segments are 4GB contiguous memory area and the starting address of the segment is aligned to 4GB). Initially, RAPID reserves memory area for a segment. The memory area corresponding to the mi-segment is made accessible when a mi-segment is allocated from a segment. The mi-segments for different heaps are allocated from different segments; thus, objects from different heap regions always belong to different segments. We also export a thread-local variable from Mimalloc that is used by the application to pass the region-id. If the caller doesn't pass the region-id, the allocation is done from the default region. We compared the performance of our region-based implementation using the default region with the original Mimalloc implementation. Our implementation slightly improved the performance of CPU SPEC 2017 benchmarks without additional memory overhead. Therefore, we have used our region-based Mimalloc implementation (that uses region-0 for every allocation) as the baseline in our evaluations.

Finding starting address. We implemented an API in Mimalloc to find the starting address of the object, which is used by the profiler. In Mimalloc, mi-segment can be derived from a virtual address using just an "and" operation due to the alignment property of mi-segments. A mi-segment also stores the metadata corresponding to every page in the mi-segment. A page contains fixed-size objects. Page metadata contains starting address of the page and the object size on that page. Our API uses page metadata to compute the starting address of the object. From a given address `x`, the location of page metadata can be efficiently computed using the offset of `x` in the corresponding mi-segment.

Stack allocations in the profile Phase. As discussed before, during the profile phase, RAPID stores the allocation site in the object header at the time of heap allocation. The compiler can't distinguish between a heap/stack location statically when a stack address escapes the static scope (i.e., by passing to a function, stored in a memory, or typecasted to an integer). In these cases, disambiguation checks may encounter stack allocations. To handle this case, if a stack address escapes the static scope, RAPID replaces the stack allocation with heap allocation. For these allocations, RAPID instruments `free` when the stack allocation goes out of scope. In disambiguation checks, if an address belongs to the data section, RAPID identifies them as global variables. For global variables, a fixed allocation site is logged in the profile phase.

Stack allocations in final code. If a disambiguation check is needed for the stack and heap location, we might need to allocate the stack object from a different region. We can do it by replacing the stack allocation with region-based heap allocation. However, this may hurt the performance because the stack allocations are faster. To handle this, we try to allocate default regions to stack allocations during region inference, thus eliminating the need to replace them. However, if a disambiguation check is needed between two stack locations, then we can't assign the default region to both of them. We found this case in three CPU SPEC 2017 benchmarks (`imagick_r`, `gcc_r` and `blender_r`). To fur-

ther minimize the overhead of stack allocation replacement, we use per-thread stacks for different regions. This is done for only those benchmarks that require stack allocations in a different region other than the default region. RAPID identifies whether per-thread stacks for different regions are needed while building the inference graph.

Handling wrappers. In some benchmarks, wrapper functions are used to allocate memory. The problem with the wrapper functions is that memory allocation in the wrapper function is treated as the allocation site for all allocations. RAPID uses an iterative algorithm, Algorithm 3, to detect the wrapper functions automatically. `FindWrappers` takes the list of all the functions in the application as an argument and returns a set of wrapper functions.

If a function does a single allocation (using standard allocation API or a wrapper function), doesn't call external functions, doesn't store addresses derived from the allocated address in memory, and returns an address derived from the allocated address, then it is a wrapper function. All functions except standard library functions are considered external functions. We found that, in the wrapper functions, library functions are used for initializing memory or handling error conditions.

`IsDerivedFrom` routine takes two pointer arguments and returns true if the first argument is derived from the second argument using some arithmetic or typecasts operations or if both pointer arguments are the same. `GetAllocatedAddr` returns the allocated address at the unique allocation site in the function. This iterative algorithm terminates when no new wrapper functions are identified during an iteration.

We could detect most wrapper functions using the algorithm described above. The `perlbench_r` and `gcc_r` benchmarks use wrapper functions (namely `Perl_my_exit`, `Perl_PerlIO_stderr`, `Perl_PerlIO_fileno`,

```
Function FindWrappers(FunctionList FL):
    /*Input: FL is the list of all the functions, Output:
     Set of wrapper functions*/
    WrapperFunctions ← ();    /*Set of wrapper functions*/
    foreach i ∈ Standard allocation API do
        WrapperFunctions.insert(i);
    end
    Changed ← true;
    while Changed do
        Changed ← false;
        foreach Function F ∈ FL do
            if F ∈ WrapperFunctions or
               NumCallsToWrapperFunctions(F) != 1 or
               NumCallsToExternalFunctions(F) > 0 then
                continue;
            end
            Ptr ← GetAllocatedAddr(F);
            StoringAllocatedAddr ← false;
            foreach Store operation S ∈ F do
                ValueStored ← S.GetValueOperand();
                if IsDerivedFrom(ValueStored, Ptr) then
                    GenerateWarning(F, Ptr);
                    StoringAllocatedAddr ← true;
                    break;
                end
            end
            if StoringAllocatedAddr then
                continue;
            end
            foreach Return operation R ∈ F do
                ReturnValue ← R.GetValueOperand();
                if IsDerivedFrom(ReturnValue, Ptr) then
                    WrapperFunctions.insert(F);
                    Changed ← true;
                    break;
                end
            end
        end
    end
    return WrapperFunctions;
```

Algorithm 3: Function `FindWrappers` takes a list of functions. It returns a subset of functions. These functions represent the list of wrapper functions in the application.

and `xexit`) for the library functions exit, stderr, and fileno in the wrapper functions for allocations. For these benchmarks, we manually added these functions to the list of library functions to detect the wrapper functions for allocations. Apart from these cases, we found that at four places in the `parest_r` benchmark, instead of returning the allocated address, the wrapper function stores the address in a class member. In the `x264_malloc` routine of the `x264_r` benchmark, the wrapper function stores the address of the allocated buffer in the buffer itself before returning the allocated address. Our algorithm generates warnings for these cases. The `GenerateWarning` routine in Algorithm 3 generates a warning if the allocated address is stored once in an address that is derived from the allocated address itself or if the allocated address is stored once in a class member. We found that with these conditions, there are only five functions for which the warning was generated, and they are indeed the wrapper functions. We expect users of our tool to inspect the routines for which warnings are generated manually.

Nevertheless, if the above algorithm misses a wrapper function, we can inspect the profiler's output to identify the wrapper functions. The profiler generates a list of conflicting accesses with the same allocation sites. In addition to the allocations inside loops, this list also contains all wrapper functions. We found that this list is usually small, which makes it easy to find the missing wrapper functions. We have also added a new function attribute in the compiler to annotate these wrapper functions manually. We also verified that we didn't miss any wrapper function by looking at the profiler's output.

Column-2 of Table 6.1 shows the total number of wrapper functions identified (including the ones for which warnings were generated) by RAPID for Polybench and CPU SPEC 2017 benchmarks using our iterative algorithm. Apart from the wrapper functions that directly allocate memory using the standard APIs, the other wrapper functions call wrapper functions to allocate memory and return the allocated address. Some of them initialize the memory buffer

Table 6.1: Wrapper functions information for Polybench and CPU SPEC 2017 benchmarks. (#WF = Total number of wrapper functions automatically identified by RAPID. The last two columns show the name and source location of the wrapper functions corresponding to the allocation sites of the conflicting memory accesses in the profiler's output. These functions are identified based on the iterative algorithm of RAPID and the profiler's output.)

Benchmark	#WF	Wrapper functions	Source Location
Polybench	1	polybench_alloc_data	polybench.c, line 557
parest_r	9	reinit	vector.h, line 973
nab_r	5	ivector	memutil.c, line 42
perlbench_r	11	Perl_safesysmalloc	util.c, line 133
gcc_r	26	xmalloc xcalloc	xmalloc.c, line 141, xmalloc.c, line 155
xalancbmk_r	7	allocate	MemoryManagerImpl.cpp, line 30
x264_r	10	x264_malloc	common.c, line 1102
blender_r	8	MEM_lockfree_mallocN MEM_lockfree_callocN	mallocn_lockfree_impl.c, line 300, mallocn_lockfree_impl.c, line 279

before returning it to the caller. Columns-3 and 4 of Table 6.1 show the subset of identified wrapper functions and their location in the source code. These functions correspond to the allocation sites of the conflicting memory accesses in the profiler's output.

During the profile phase and final code generation, calls to wrapper functions are treated as allocation sites. As discussed earlier, our region-based allocator receives the region-id in a thread-local variable. During the final code generation, RAPID stores the region-id in the thread-local variable before calling a wrapper function that is eventually passed to the region-based allocator. During the profile phase, RAPID stores the unique identifier for the allocation site in the thread-local variable before calling the wrapper function. The allocator stores the allocation site passed by the caller (using the thread-local variable) in the object header.

6.4 Evaluation

We evaluated RAPID on an Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz 12-core machine with x86_64 architecture, 32 GB primary memory, and the Ubuntu

20.04.1 LTS operating system. We disabled hyper-threading during the evaluation. We used 46 micro and real-world benchmarks for evaluation. These include 30 Polybench (v4.2.1) and 16 C/C++ CPU SPEC 2017 benchmarks. We used the default input set for Polybench (i.e., large input set) and reference input set for CPU SPEC 2017 benchmarks in our evaluation. In our experiments, for all benchmarks, we reported the arithmetic mean of the execution times for five runs. We obtained memory overheads by computing the percentage change in the value of "maximum resident set size" using "/usr/bin/time -v" command. We used "O3" optimization level to compile the application in both phases. We used the same input sets used for profiling to evaluate the performance of our scheme. We also computed the worst-case overhead of our scheme. In the worst-case, the information gathered during the profile phase never holds at runtime. To compute the worst-case overhead, we insert the dynamic checks in a way that the last check in the loop-versioning condition never holds. Consequently, after executing disambiguation checks for all conflicting pointer pairs in a loop versioning condition, the program always takes the unoptimized path.

6.4.1 Polybench Benchmarks

We now discuss the results for Polybench benchmarks. We obtained the performance benefits by computing the percentage decrease in the execution time of these benchmarks. The positive and negative numbers for performance benefits represent the improvement and degradation over the native execution time of the benchmark, respectively.

At first, we compiled and executed these benchmarks with `restrict` keyword with option `POLYBENCH_USE_RESTRICT`, provided by Polybench. This option attaches `restrict` keyword to the function arguments. Users can annotate the function arguments with `restrict` to provide additional aliasing information to the compiler. Compilers trust these annotations during code generation. The `restrict` keyword informs the compiler that the pointer does not alias with any other pointer. We performed this experiment to ensure that RAPID

Table 6.2: Polybench performance benefits for `restrict` keyword Scout and RAPID. (#R_r = Number of regions identified by RAPID, #L_v = Number of loops versioned, P_b = Performance benefits (in percentage), P_{wr} = Worst case performance for RAPID when the dynamic checks always fail excluding the allocator's overhead (in percentage). Positive and negative numbers for performance benefits and worst case performance represent improvement and degradation, respectively.)

	Restrict	**Scout**		**RAPID**			
Benchmark	P_b	#L_v	P_b	#R_r	#L_v	P_b	P_{wr}
gesummv	46.04	1	49.37	3	1	47.58	-0.1
2mm	3.88	2	4.22	2	2	3.36	-0.09
3mm	4.63	3	4.28	2	3	4.3	-0.09
bicg	53.32	1	51.11	3	1	52.94	-0.07
doitgen	10	1	10.2	2	1	4.81	-0.05
jacobi-1d	-1.27	2	11.8	2	2	5.39	-0.12
jacobi-2d	3.43	2	2.37	2	2	3.31	-0.1

does not miss out any optimization opportunities enabled by the user annotation using `restrict`. Column-2 of Table 6.2 shows the performance benefits of these benchmarks using `restrict`. This table also shows the results obtained when compiled with Scout [111] and RAPID. Column-3 and Column-6 show the number of versioned loops with Scout and RAPID. Column-4 and Column-7 show the performance benefits obtained using Scout and RAPID. Column-5 shows the number of regions needed for these benchmarks in our approach. Column-8 shows our worst-case performance, which is essentially the overheads of our dynamic checks. Notice that in the worst case, the program always takes the unoptimized path after executing the dynamic checks. Table 6.2 shows the benchmarks with more than 3% performance benefits in execution time with RAPID.

For four out of seven benchmarks, namely, `gesummv`, `2mm`, `3mm`, and `bicg`, the performance of RAPID, `restrict`, and Scout are similar. We observed that the `restrict` annotation slowed down the performance of `jacobi--1d`. Our implementation of Scout exhibited similar behavior for this benchmark. This is because `jacobi-1d` failed to preserve the non-aliasing annotations across the optimization passes, as also observed using Scout.

For `jacobi-1d`, the improvement with RAPID is lesser than Scout, whereas, for `jacobi-2d` RAPID performs better than Scout. We observed that the reason behind this is the different allocators used in these benchmarks. Scout used Jemalloc allocator. For `jacobi-1d`, Mimalloc performs better than Jemalloc, and for `jacobi-2d`, Jemalloc performs better than Mimalloc. To further investigate this, we ran RAPID with Jemalloc. For this experiment, we modified the loop-versioning condition to check if two addresses are different (this condition is always true) rather than checking if the top 32-bits are different. Notice that the original loop-versioning condition is not valid for Jemalloc because it doesn't know anything about regions. In this experiment, RAPID reported 11.92% and 2.68% performance benefits for `jacobi-1d` and `jacobi-2d`, respectively. Our implementation of Scout exhibited similar improvements.

The performance benefit for `doitgen` with RAPID is less than using `restrict` keyword and Scout. We found that during compilation with RAPID, loop invariant code motion moved instructions to perform "xor" operations and checking of top 32-bits to the outermost loop. This caused register pressure during the register allocation, and a local variable was spilled resulting in additional memory accesses inside loop. To verify this, we disabled loop invariant code motion after inserting the dynamic checks. We obtained performance benefits of around 9.02% for this benchmark similar to `restrict` keyword and Scout.

We observed that the slowdown introduced by the dynamic checks (Column-8 of Table 6.2) was always less than 0.13% for these six benchmarks. Moreover, the slowdown was always less than 0.3% for all the 30 Polybench benchmarks. These results indicate that the overheads of our dynamic checks are not high for the Polybench benchmarks.

We compared our results with the hybrid approach (discussed in [24]). The hybrid approach uses both polyhedral [71–73] and symbolic range [74–76] analy-

Table 6.3: Polybench speedups using `restrict` keyword, hybrid approach [24], Scout [111] and RAPID. (S_{r3} = Speedup using `restrict` keyword with LLVM-3.6.0, S_{r10} = Speedup using `restrict` keyword with LLVM-10, S_h = Speedup using hybrid approach, S_s = Speedup using Scout, S_r = Speedup using RAPID.)

Benchmark	S_{r3}	S_{r10}	S_h	S_s	S_r
gesummv	1.86	1.86	2.5	1.98	1.91
bicg	2.14	2.14	2.7	2.05	2.13
gramschmidt	1.01	1	1.4	1.01	1.01

ses to compute the range based dynamic checks in $O(1)$. Table 6.3 shows the speedups for three benchmarks that resulted in substantial performance benefits using the hybrid approach [24]. We found that the benefits reported by the hybrid approach is better than our tool. Since the hybrid approach uses LLVM-3.6.0, we performed another experiment using `restrict` keyword with LLVM-3.6.0 to estimate the runtime of these benchmarks using the hybrid approach on our machine. Table 6.3 shows the speedups obtained when `restrict` keyword is used with LLVM-3.6.0 (Column-2), `restrict` keyword with LLVM-10 (Column-3), the hybrid approach [24] (Column-4), Scout [111] (Column-5), and RAPID (Column-6). We observed that these benchmarks show similar speedups as that of Scout and `restrict` keyword with RAPID. We believe that the performance improvement is lesser than the hybrid approach due to the different processor versions.

For six benchmarks, namely, `trmm`, `durbin`, `trisolv`, `floyd-warshall`, `nussinov` and `seidel-2d`, RAPID failed to insert the dynamic checks. Five of these benchmarks perform load/store to the same array inside the loop, and the remaining benchmark did not show any potential to be optimized further. We also observed a similar behavior in Scout. We further verified that the enabled set of optimizations for all of Polybench benchmarks was similar to that of Scout by inspecting the IR.

Performance improvements for 23 benchmarks (other than those shown in Table 6.2) is in the range of 0% to 2.52%. The maximum number of regions

needed for these benchmarks is three. Our approach enables optimization opportunities such as loop invariant code motion, global value numbering, store elimination, and efficient vectorization for these benchmarks, leading to performance benefits.

For Polybench benchmarks, the memory overhead of our custom allocator lies in the range of -0.68% to 0.09%, with a geometric mean of 0.03%. The CPU overhead lies in the range of -0.73% to 0.5%, with a geometric mean of 0.03%. Scout exhibited CPU overheads in the range of -2.55% to 5.3% for Polybench benchmarks. The geometric mean CPU overhead in our approach is better than the 0.21% geometric mean overhead observed using Scout. The absolute memory overhead for Polybench benchmarks is always less than 1% for both Scout and RAPID.

Overall, the results for the PolyBench benchmarks show that RAPID can enable performance improvements when loop versioning exposes additional optimization opportunities, while introducing very low runtime overhead. In many cases, the achieved performance is comparable to using the `restrict` keyword or Scout, and the overhead from dynamic checks remains negligible. The results also indicate that performance gains depend on the extent to which loop versioning is applicable and preserved across compiler optimizations.

6.4.2 CPU SPEC 2017 Benchmarks

We now discuss the results for CPU SPEC 2017 benchmarks. We use two metrics throughout this section,

1. Overhead represents the percentage increase. Positive and negative numbers represent the degradation and improvement, respectively.
2. Performance benefits represent the percentage decrease in the benchmark's execution time. Positive and negative numbers represent the improvement and degradation in the execution time, respectively.

Table 6.4: CPU and memory overhead for CPU SPEC 2017. (CO_S = CPU overhead of the Scout's allocator (in percentage), CO_R = CPU overhead of the RAPID's allocator (in percentage), MO_S = Memory overhead of the Scout's allocator (in percentage), MO_R = Memory overhead of the RAPID's allocator (in percentage). Positive and negative numbers for overhead represent degradation and improvement, respectively.)

Benchmark	CO_S	CO_R	MO_S	MO_R
namd_r	0.51	0.08	-2.22	0.11
parest_r	1.15	0.39	11.79	1.59
imagick_r	0.37	0.36	-0.32	-0.02
nab_r	0.73	-0.33	17.6	1.07
perlbench_r	8.55	0.31	6.46	6.96
gcc_r	1.22	-0.36	5.67	-0.62
omnetpp_r	14.08	0.14	16.81	-0.03
xalancbmk_r	3.36	-1.3	19.99	3.99
x264_r	-1.93	0.27	0.08	0.17
deepsjeng_r	2.41	0.18	0.06	-0.01
povray_r	11.87	0.13	36.79	5.46
blender_r	1.78	-2.27	5.25	1.17
GM	**3.01**	**0.2**	**7.54**	**1.62**

CPU and memory overhead of the allocator. We first discuss our region-based allocator's CPU and memory overheads, shown in Table 6.4. Column-3 and Column-5 show allocator's CPU and memory overheads for RAPID, respectively. We obtained the CPU overheads by computing the percentage change in the execution times of the benchmarks when the allocator always allocates memory in the default region and when the allocator allocates memory in the regions identified by the `Region Inference` (Section 6.2). We disabled loop-versioning in this experiment.

The CPU overhead of RAPID's allocator is always less than 0.5% with a geometric mean of 0.2%, shown in Table 6.4. The memory overhead of RAPID's allocator is always less than 7% with a geometric mean of 1.62%. This overhead is more than 3% for only three out of 12 benchmarks. Mimalloc allocates at least one `mi-segment` (a large contiguous area) for every heap region. `mi-segment` is further divided into pages that are used to allocate objects of different sizes. In the case of multiple regions, Mimalloc needs to allocate

at least one `mi-segment` for every heap region. Multiple `mi-segments` are adding additional memory overhead in some benchmarks. For `xalancbmk_r` and `blender_r` benchmarks, the multi-region allocation approach improved the performance by 1.3% and 2.27%, respectively. We believe that this might be due to the presence of multiple per-heap allocator caches with different regions. Scout achieved similar speedup for the `x264_r` benchmarks.

Additionally, this table shows the CPU (Column-2) and memory (Column-4) overheads of the allocator used by Scout [111]. Scout reported 3.36%, 8.55%, 11.87% and 14.08% CPU overheads for `xalancbmk_r`, `perlbench_r`, `povray_r`, and `omnetpp_r`. Due to the high overheads, Scout can't be used for these benchmarks despite some additional loops being versioned. Apart from these, Scout reported more than 3% overheads for four benchmarks, which was more than the benefits of loop-versioning in these benchmarks. On the other hand, due to the low overhead of RAPID's allocator, we could improve the performance of all these benchmarks. The memory overheads in Scout is also substantial. For five benchmarks, the memory overhead in Scout's allocator is more than 10%. For these benchmarks, the memory overhead of RAPID's allocator is always less than 6%. The primary reason for the high memory overhead in Scout's allocator is the additional padding required to satisfy constraints on allocation size and object alignment. RAPID doesn't change the allocation size and alignment of the objects. Based on the results, we can conclude that RAPID's allocator outperforms the allocator used by Scout for most of CPU SPEC 2017 benchmarks.

Performance benefits. We now discuss the performance benefits for the CPU SPEC 2017 benchmarks. We compared the performance benefits obtained using RAPID with Scout. Table 6.5 shows the results for these benchmarks using RAPID and Scout. Column-4 shows the number of regions needed for the benchmarks. Column-5 shows the number of versioned loops. Column-2 and Column-6 show the performance benefits excluding the allocator's overhead.

Table 6.5: CPU SPEC 2017 performance benefits for RAPID and Scout. (#R_r = Number of regions required, #L_v = Number of loops versioned, P_b = Performance benefits over native excluding the allocator's overhead (in percentage), P_{ba} = Performance benefits over native including the allocator's overhead (in percentage), P_{ws} = Performance benefits over native excluding the allocator's overhead for Scout in the absence of the profiler (in percentage), P_{wr} = Worst case performance for RAPID when the dynamic checks always fail excluding the allocator's overhead (in percentage). Positive and negative numbers for performance benefits and worst case performance represent improvement and degradation, respectively.)

	Scout w/ Profiler		RAPID				Worst case	
Benchmark	P_b	P_{ba}	#R_r	#L_v	P_b	P_{ba}	P_{ws}	P_{wr}
namd_r	0.51	0	2	9	0.15	0.08	-0.38	-0.08
parest_r	0.73	-0.42	4	68	0.43	0.04	-0.3	-0.04
imagick_r	0.23	-0.14	2	4	0.49	0.14	0	0
nab_r	0.27	-0.46	4	21	0.11	0.44	-0.09	-0.49
perlbench_r	0	-8.55	3	12	0.61	0.31	-0.4	0
gcc_r	0.73	-0.49	4	35	0.64	1	0.13	0
omnetpp_r	0.5	-13.58	2	6	0.74	0.6	-1.79	-0.05
xalancbmk_r	1.47	-1.89	3	34	1.69	2.97	0.88	-0.47
x264_r	0.89	2.82	3	32	1.11	0.85	0	-0.07
deepsjeng_r	0.49	-1.92	2	6	0.6	0.42	-0.1	-0.18
povray_r	0	-11.87	2	2	1.72	1.6	-2.11	-0.17
blender_r	0.52	-1.26	4	30	0.9	3.14	-3.37	-0.42

Column-3 and Column-7 show the overall performance benefits, including the allocator's overhead. The overall performance benefits (Column-3) for Scout are approximate and computed based on the allocator's overhead and performance benefits observed during the experiments.

Table 6.5 shows results for 12 benchmarks out of 16. The remaining four benchmarks did not show any performance benefits due to the following reasons,

1. None of the loops shows potential to be optimized further (`lbm_r`, `mcf_r`).
2. Versioned loops never get executed (`leela_r`, `xz_r`) during the profile phase.

For comparison, we took the maximum improvement observed using Scout for each benchmark across all configurations. Scout reported a performance improvement in the range of 0.23% to 1.47% (Column-2 in Table 6.5) for ten benchmarks, excluding the allocator overhead. However, after considering the allocator overhead, only one benchmark showed better performance (Column-3). RAPID improved the performance of all 12 benchmarks in which the versioned loops execute at runtime. After excluding the allocator overhead, the benefits are in the range of 0.11% to 1.72% (Column-6). The overall improvements (including allocator overhead) for these 12 benchmarks are in the range of 0.04% to 3.14% (Column-7). For some benchmarks, the overall improvement is higher because RAPID's allocator performs better than the native in these cases. From these results, we conclude that because of the low overhead of the allocator, RAPID can improve the performance of a larger set of benchmarks than Scout.

Columns-4 and 5 in Table 6.5 show the number of regions and versioned loops for these benchmarks. The maximum number of regions needed for these benchmarks is four. In this experiment, we didn't version loops that require checks for objects allocated at the same site.

Column-9 of Table 6.5 shows the worst-case performance of CPU SPEC 2017 benchmarks with RAPID. Scout uses a profiler to filter loops that don't show improvement at runtime. We compare RAPID's worst-case performance with the performance of Scout without the profiler (Column-8 of Table 6.5). The performance degradation ranges from 0.09% to 3.37% and 0.04% to 0.49% for Scout and RAPID, respectively. Based on the results, the slowdown introduced by Scout in the absence of the profiler could be as high as 3.37%. On the other hand, the slowdown introduced by RAPID was always less than 0.5%, even when the dynamic checks always led to a failure. We can conclude that even in the worst-case scenario, the maximum slowdown introduced by RAPID is not substantial and less than Scout.

Table 6.6: CPU SPEC 2017 performance benefits handling memory allocations inside the loops. (#L_v = Number of loops versioned, P_{wr} = Worst case performance for RAPID across all configurations when the dynamic checks always fail excluding the allocator's overhead (in percentage), T_{reg} = Threshold for maximum number of regions available, P_b = Performance benefits over native excluding the allocator's overhead (in percentage), CO_R = CPU overhead of the RAPID's allocator (in percentage), MO_R = Memory overhead of the RAPID's allocator (in percentage). Positive and negative numbers for performance benefits and worst case performance represent improvement and degradation, respectively.)

			T_{reg} = 8			T_{reg} = 16			T_{reg} = 32		
Benchmark	#L_v	P_{wr}	CO_R	MO_R	P_b	CO_R	MO_R	P_b	CO_R	MO_R	P_b
parest_r	69	-0.25	0.18	2.8	-0.08	0.25	3.85	0.15	0.46	5.14	-0.08
xalancbmk_r	34	-0.48	-1.76	4.06	-0.19	-1.67	4.11	-0.38	-1.67	4.22	-0.48
blender_r	31	-0.37	-3.67	1.98	-0.25	-3.38	2.25	-0.19	-2.91	3.01	0.3

Table 6.7: CPU SPEC 2017 performance benefits using our loop filtering mechanism with the threshold for number of iterations as 64. (#L_v = Number of loops versioned, T_{reg} = Threshold for maximum number of regions available, P_b = Performance benefits over native excluding the allocator's overhead (in percentage), P_{ba} = Performance benefits over native including the allocator's overhead (in percentage). Positive and negative numbers for performance benefits represent improvement and degradation, respectively.)

	T_{reg} = 0				T_{reg} = 8		T_{reg} = 16		T_{reg} = 32	
Benchmark	#L_v	P_b	P_{ba}	#L_v	P_b	P_{ba}	P_b	P_{ba}	P_b	P_{ba}
parest_r	58	0.43	0.04	59	0.08	-0.11	0.18	-0.08	0.39	-0.08
xalancbmk_r	14	1.88	3.15	14	0.95	2.69	0.85	2.5	1.7	3.34
blender_r	29	1.19	3.43	30	0.07	3.72	0.19	3.55	0.54	3.43

We found that in three benchmarks `parest_r`, `xalancbmk_r` and `blender_r`, the allocation sites for conflicting accesses are the same. As discussed in (Section 6.2.3.1), for conflicting allocation sites, we ask the user to provide the maximum number of regions at compile time that is distributed equally among the conflicting allocation sites after assigning regions to non-conflicting allocation sites. We executed these benchmarks with the maximum number of regions 8, 16, and 32. Table 6.6 shows the results of this experiment. Column-2 shows the number of versioned loops. Column-3 shows the worst-case performance when the dynamic checks always fail. Column-4, 7, and 10 show the CPU overhead of the allocator for the different regions. Column-5, 8, and 11 show the memory overhead. Column-6, 9, and 12 show the performance benefits af-

ter excluding the allocator overhead. In `xalancbmk_r`, objects with conflicting and non-conflicting allocation sites are accessed at different times. Therefore, the number of versioned loops in Table 6.5 and Table 6.6 are the same.

As discussed, we allocate region-ids for these allocation sites from a set of ids in a round-robin manner in the hope that the conflicting accesses would belong to different regions. As we increase the number of regions, the probability of dynamic check failure decreases. For `parest_r`, with eight regions, the dynamic checks failed 454792 times out of 67025062 (0.67%). However, none of the checks failed with 16 and 32 regions. Similarly, for `xalancbmk_r`, the checks failed for 90 out of 252267 (0.03%) times with our default approach. This number was reduced to 16 (0.01%) with eight regions. For 16 and 32 regions, none of the checks failed. For `blender_r`, none of the dynamic checks failed for 8, 16, and 32 regions. After handling conflicting allocation sites, RAPID could version all the loops versioned by Scout that are executed at runtime. The maximum slowdown introduced by RAPID in the worst-case scenario while versioning all the loops for `parest_r`, `xalancbmk_r` and `blender_r` is still less than 0.5% (Column-3 of Table 6.6).

As shown in Table 6.6, the CPU and memory overheads of the allocator increase with the number of regions. This is due to the additional cost of allocating and accessing memory in multiple regions. Despite the degradation, the CPU and memory overhead is always less than 0.5% and 6%, respectively. The performance degraded (Columns-6, 9, 12) for all of these benchmarks when we enabled loop-versioning (excluding the allocator overhead). Further investigation revealed that versioning some loops with fewer iterations resulted in the generation of relatively unoptimized code for the rest of the function.

To improve further, we use a loop filtering mechanism. We disable loop-versioning for loops with the total number of iterations is less than a specific threshold. Table 6.7 shows the performance benefits for the different number of

regions and loop filtering threshold 64. The zero number of regions is the case when we don't handle the conflicting allocation sites (as in Table 6.5). All the benchmarks showed positive improvement after excluding the allocator overhead (Columns- 3, 6, 8, 10). However, the overall performance of `parest_r` was negative (Columns- 7, 9, 11) due to allocator overhead. The performance of `xalancbmk_r` further improved from 1.69% (Table 6.5) to 1.88% when we filtered loops with fewer iterations.

We performed the same experiment for other benchmarks as well. We also experimented with different thresholds for loop filtering. However, there was no noticeable improvement over the benefits shown in Table 6.5. We can conclude that wisely filtering out the loops with small iterations can also improve the performance of the benchmarks.

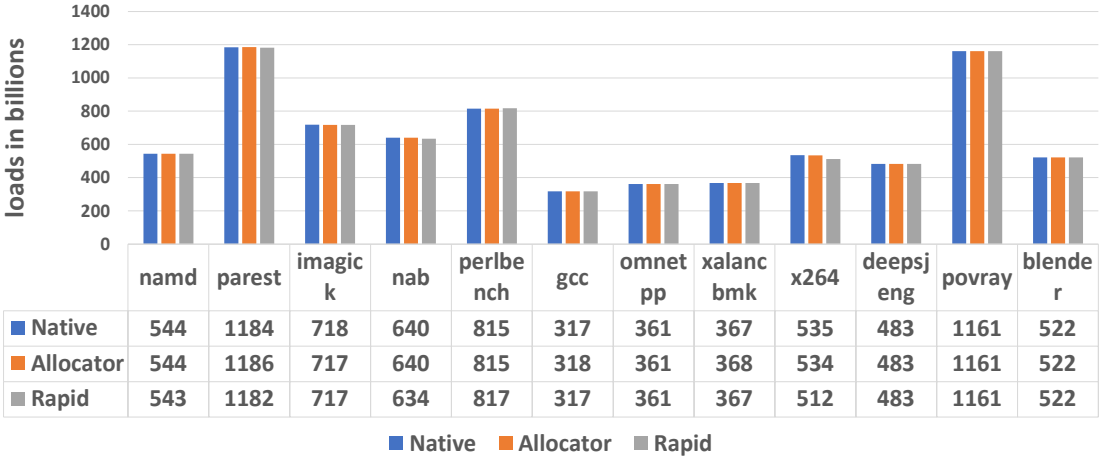
Impact of Data and Instruction Cache. To understand the impact of the change in the memory layout, we computed the number of L1 data cache loads as shown in Figure 6.6. The first, second, and third columns correspond to the baseline, RAPID's allocator without dynamic checks, and RAPID, respectively. As expected, we didn't observe any significant change in the number of loads due to the allocator. However, when the dynamic checks are enabled, we observed a reduction in the number of loads for `nab_r` and `x264_r`. This is due to additional loop-vectorization and loop invariant code motion (LICM) optimizations enabled by RAPID. The vectorized code can load a 16-byte value using a single instruction. Such loads are counted as a single L1 cache load. The corresponding non-vectorized loop may use four 4-byte reads or two 8-byte reads to read 16 bytes, which are counted as four and two L1 cache loads, respectively. The LICM optimization reduces the number of L1 cache loads because the load is moved outside the loop. Figure 6.7 shows the percentage miss in the L1 data cache. The cache misses are slightly low for the `xalancbmk_r` benchmark with RAPID's allocator. We believe that this is due to the change in the memory layout. For `xalancbmk_r` benchmark, the RAPID's allocator also

performs better than the baseline as shown in Table 6.4. For other benchmarks, the percentage of cache misses using RAPID's allocator are almost similar.

For `parest_r`, `xalancbmk_r`, and `blender_r`, we also computed the number of L1 data cache loads and percentage miss for 256 regions. The number of loads in billions and percentage misses for these benchmarks using RAPID's allocator were 1183 (9.79), 366 (10.84), and 522 (2.36), which is not very different from the numbers for small regions. Even with the dynamic checks, the numbers are similar for 256 regions.

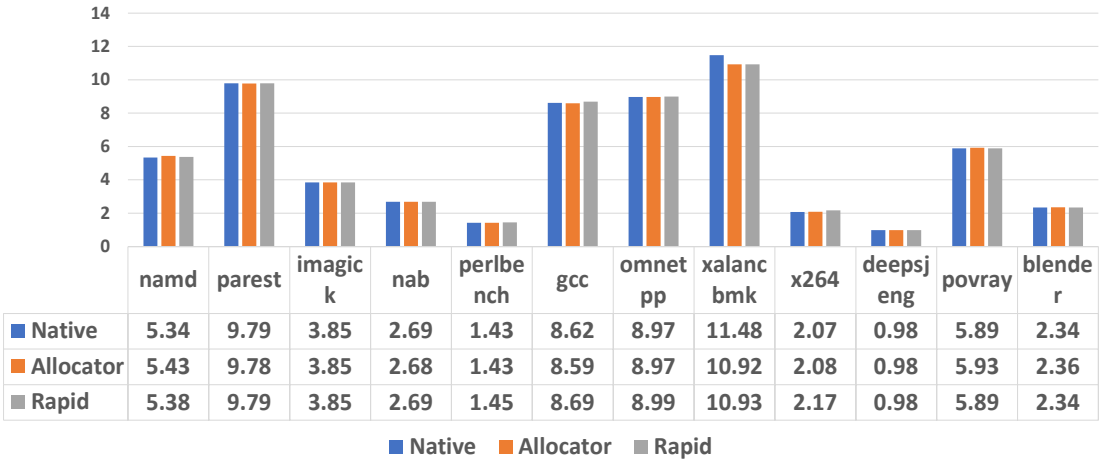
To understand the impact of the change in the code layout, we computed the number of L1 instruction cache loads as shown in Figure 6.8. The first, second, and third columns correspond to the baseline, RAPID's allocator without dynamic checks, and RAPID, respectively. Except `imagick_r` and `blender_r`, we didn't observe any significant change in the number of loads due to the allocator. Both these benchmarks use per-thread stacks because of the conflicting stack allocations. Allocation and deallocation from per-thread stacks add additional instructions that change the behavior of the instruction cache. With dynamic checks, the number of instruction cache loads varies for more benchmarks because of the optimizations enabled for the loop. The percentage miss in the instruction cache misses is shown in Figure 6.9. The cache misses reduced significantly for `deepsjeng_r` benchmarks using dynamic checks. The number of instruction cache loads was also reduced from 503 billion to 493 billion for this benchmark with dynamic checks (Figure 6.8). We believe this might be due to the reduced code size of the optimized version of the loop. The overall increase in the code size is negligible for most benchmarks except for `nab_r` (1.09%) and `deepsjeng_r` (3.1%).

For `parest_r`, `xalancbmk_r`, and `blender_r`, we also computed the number of L1 instruction cache loads and percentage miss for 256 regions. The number of loads in billions and percentage miss for these benchmarks using



	namd	parest	imagick	nab	perlbench	gcc	omnetpp	xalancbmk	x264	deepsjeng	povray	blender
Native	544	1184	718	640	815	317	361	367	535	483	1161	522
Allocator	544	1186	717	640	815	318	361	368	534	483	1161	522
Rapid	543	1182	717	634	817	317	361	367	512	483	1161	522

Figure 6.6: Number of loads (in billions) in data cache (L1) for native, allocator, and RAPID.



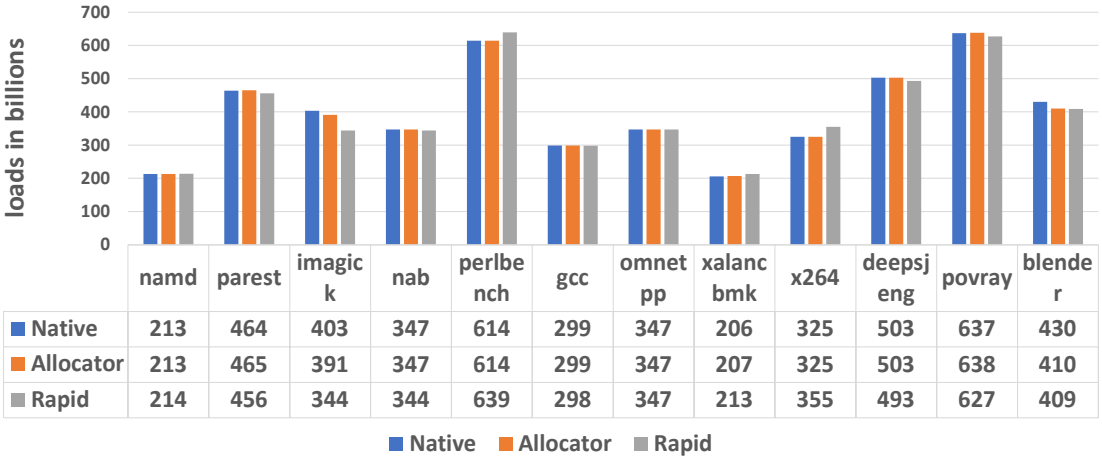
	namd	parest	imagick	nab	perlbench	gcc	omnetpp	xalancbmk	x264	deepsjeng	povray	blender
Native	5.34	9.79	3.85	2.69	1.43	8.62	8.97	11.48	2.07	0.98	5.89	2.34
Allocator	5.43	9.78	3.85	2.68	1.43	8.59	8.97	10.92	2.08	0.98	5.93	2.36
Rapid	5.38	9.79	3.85	2.69	1.45	8.69	8.99	10.93	2.17	0.98	5.89	2.34

Figure 6.7: Percentage miss in data cache (L1) for native, allocator, and RAPID.

RAPID's allocator were 465 (0.07), 207 (1.05), and 409 (0.28), which is not very different from the numbers for small regions. For 256 regions, the numbers were similar with dynamic checks.

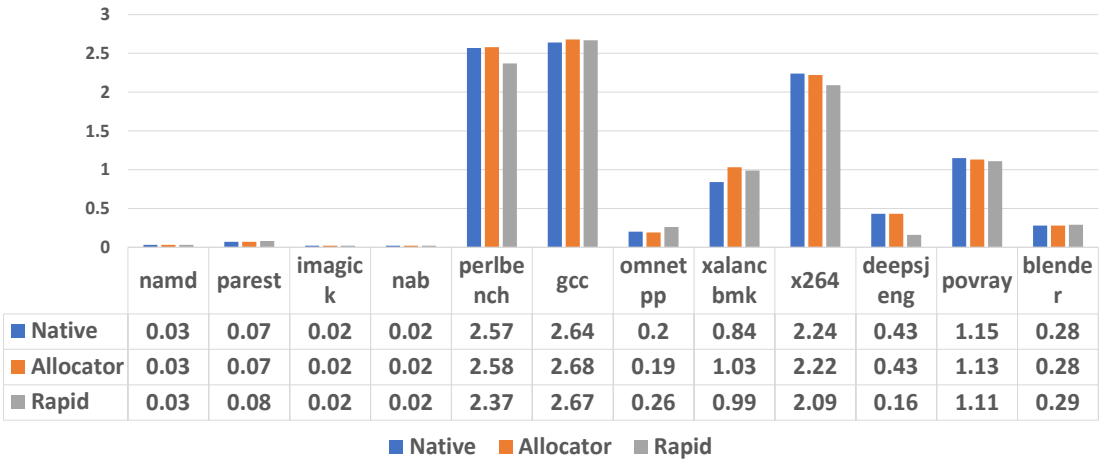
Profiler overhead. The overhead of the profiler includes the following:

1. Replacing stack allocations with Mimalloc allocation API and `free` (Section 6.3).
2. Finding object headers at runtime (Section 6.3) to identify objects and their allocation sites.
3. Storing the loop-related information to the log file (Section 6.2).



	namd	parest	imagick	nab	perlbench	gcc	omnetpp	xalancbmk	x264	deepsjeng	povray	blender
Native	213	464	403	347	614	299	347	206	325	503	637	430
Allocator	213	465	391	347	614	299	347	207	325	503	638	410
Rapid	214	456	344	344	639	298	347	213	355	493	627	409

Figure 6.8: Number of loads (in billions) in instruction cache (L1) for native, allocator, and RAPID.



	namd	parest	imagick	nab	perlbench	gcc	omnetpp	xalancbmk	x264	deepsjeng	povray	blender
Native	0.03	0.07	0.02	0.02	2.57	2.64	0.2	0.84	2.24	0.43	1.15	0.28
Allocator	0.03	0.07	0.02	0.02	2.58	2.68	0.19	1.03	2.22	0.43	1.13	0.28
Rapid	0.03	0.08	0.02	0.02	2.37	2.67	0.26	0.99	2.09	0.16	1.11	0.29

Figure 6.9: Percentage miss in instruction cache (L1) for native, allocator, and RAPID.

The slowdown for the CPU SPEC 2017 lies in the range of 1.1% (`nab_r`) to 751.87% (`povray_r`). We noticed that replacing stack allocations with Mimalloc allocation API and `free` contributed significantly to the slowdown for these benchmarks. Replacing stack allocations with the Mimalloc allocation API and `free` introduced slowdowns ranging from 0.55% (`nab_r`) to 251.87% (`povray_r`). The slowdown lies between 0.55% (`nab_r`) to 50.28% (`x264_r`), excluding the overhead incurred due to replacing stack allocations.

Overall, the results for the CPU SPEC 2017 benchmarks show that RAPID maintains low allocator overhead while enabling performance improvements across a broader set of benchmarks compared to prior approaches. The region-based allocator incurs small CPU and memory overheads, which allows versioned loops to translate into net performance gains in many cases. Even when dynamic checks fail, the worst-case slowdown remains low, demonstrating predictable behavior. Additional mechanisms such as profiling, region tuning, and loop filtering further help identify profitable loops and reduce unnecessary overhead. Together, these results indicate that RAPID provides a practical and robust approach for improving performance on real-world applications.

6.5 Summary

This chapter presented RAPID, a profiling-guided tool that combines loop-versioning with constant-time region-based dynamic checks to enable precise and scalable pointer disambiguation at runtime. Unlike prior approaches that impose significant CPU or memory overheads due to their allocator designs or dynamic check mechanisms, RAPID uses profiling to selectively identify memory objects that need disambiguation and allocates them in distinct regions. This targeted allocation strategy minimizes runtime overhead while retaining the benefits of high-precision alias information. Our evaluation showed that the number of regions required across real-world benchmarks is small, making the region-based allocator highly efficient. RAPID enabled a wide range of optimizations, including load and store elimination, loop-invariant code motion, and vectorization by

```
//nab_r
//sff.c 1193
//Improvement 29%
//prm is a pointer
//to structure.
for(i = 0;
    i < prm->Natom;
    i++){
  pairlistnp[i]
    = NULL;
  lpairsnp[i] =
    upairsnp[i] = 0;
}
```

```
//nab_r
//prm.c 1217
//Improvement 30%
//prm is a pointer
//to structure.
for(i = 0;
    i < prm->Natom;
    i++) {
  if(i + 1 ==
    prm->
    Ipres[res + 1])
    res++;
  prm->AtomRes[i]
    = res;
}
```

```
//xalancbmk_r
//ElemStack.cpp 235
//Improvement 42%
//curRow is a pointer
//to structure.
for (; index < curRow
    ->fChildCount;
    index++)
  newRow[index] =
    curRow->
    fChildren[index];
```

```
//xalancbmk_r
//BaseRefVectorOf.c
//282
//Improvement 69%
//fCurCount is a
//class member.
for (; index <
    fCurCount;
    index++)
  newList[index] =
    fElemList[index];
```

```
//xalancbmk_r
//ValueVectorOf.c
//247
//Improvement 29%
//fCurCount is a
//class member.
for(unsigned int
    index = 0;
    index < fCurCount;
    index++)
  newList[index] =
    fElemList[index];
```

```
//x264_r
//macroblock.c 388
//Improvement 40%
//h is a pointer
//to structure.
for( int i = 0;
    i < h->_ref1;
    i++ )
  h->fdec->
    ref_poc[1][i] =
    h->fref1[i]->
    i_poc;
```

Figure 6.10: Loops showing a performance improvement of more than 20% from CPU SPEC 2017 benchmarks in addition to the code snippets shown by Scout in Chapter 5.

providing the compiler with accurate aliasing information within performance-critical loops. As a result, the tool achieved measurable performance improvements across all 12 CPU SPEC 2017 benchmarks where the versioned loops were executed. These findings demonstrate that RAPID is both effective and scalable for real-world applications. As future work, our approach can be extended to support function-level pointer disambiguation that could enable even more aggressive interprocedural optimizations.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis explored how constrained memory allocation can be strategically employed to perform efficient runtime pointer disambiguation, enabling precise alias information in performance-critical regions of code. We demonstrated that traditional static alias analysis used in production compilers, though scalable, often sacrifices precision, resulting in missed optimization opportunities. To bridge this gap, our first contribution introduced a profiling-based framework to estimate the performance impact of improved alias precision. This framework helped identify missing alias information and the specific optimizations that could benefit from improved precision. Based on these potential improvements, we concluded that there is a need for more precise yet scalable alias analysis implementations in production compilers.

Building on these insights, we proposed two scalable runtime techniques to enable loop-level optimizations by increasing alias analysis precision. The first approach used a segment-based allocator that enabled constant-time disambiguation checks using a single memory access by constraining the size and alignment of all memory allocations. However, it also introduced CPU and memory overheads due to the constraints on all allocations, which limited its scalability for certain real-world applications.

To mitigate this overhead, our second approach selectively constrained memory allocations by assigning potentially conflicting allocations to different regions. This region-based custom allocator maintained constant-time disambiguation, avoided memory access during checks, and significantly reduced overhead. Thus, this approach successfully scaled across all benchmarks we evaluated, demonstrating its practical viability for real-world applications. However, its primary limitation lies in the need for a profiling step to identify which allocations require disambiguation checks, introducing additional runtime costs during profiling.

Therefore, both techniques offer complementary trade-offs: the segment-based allocator avoids the need for profiling and is simpler to implement but constrains all memory allocations, resulting in higher allocator overhead and limited scalability. In contrast, the region-based allocator enables constant-time disambiguation without memory access and improves scalability, but introduces additional complexity due to its reliance on profiling.

Together, these contributions show that aggressive compiler optimizations reliant on precise alias analysis, such as vectorization, loop-invariant code motion, and dead store/load elimination, can be made scalable for real-world applications through careful design of memory allocation strategies. By enabling constant-time pointer disambiguation through constrained allocation, our techniques allow compilers to safely apply optimizations that are otherwise blocked by conservative alias assumptions. This not only improves runtime performance but also broadens the applicability of advanced compiler transformations that were previously limited to idealized or restricted scenarios.

7.2 Future Work

This dissertation lays a foundation for scalable and precise alias analysis through profiling and constrained memory allocation, opening up several promising directions for future research. Three such avenues include: (1) selecting

alias annotations based on their practical impact on optimizations and performance, (2) extending Horizon+ to be input-agnostic using symbolic analysis, and (3) extending pointer disambiguation to support interprocedural optimizations.

7.2.1 Selecting alias annotations based on their practical impact on optimizations and performance

Alias annotations can have widely varying effects on the performance of optimized code. While some annotations directly enable impactful optimizations, others may lead to optimizations that end up degrading performance. Treating all alias annotations as equally important can introduce unnecessary overhead without proportional benefit. To address this, a data-driven approach can be employed to learn which alias annotations have the highest payoff.

A supervised learning model can be trained using profiling data collected from running multiple programs with different alias annotations (obtained using Horizon+). Features such as loop structure, memory access patterns, control-flow complexity, and cache behavior can be captured to correlate alias annotations with resulting performance improvements. This model can help choose annotations that consistently unlock useful optimizations. Ultimately, this would enable the compiler to focus its resources on the annotations that actually matter for performance, improving scalability while preserving the practical value of alias information.

7.2.2 Symbolic Analysis for Input-Agnostic Annotations

Horizon+ captures aliasing behavior based on specific program inputs. While effective in practice, this limits its ability to reason about unseen inputs. To overcome this, we can augment the profiling phase with symbolic analysis techniques. Symbolic analysis allows pointer and control-flow behaviors to be represented as symbolic constraints, enabling generalization across different inputs and execution paths.

By integrating symbolic reasoning with dynamic profiling, it may be possible to infer alias relationships that are provably valid across most feasible inputs, thereby reducing dependence on input-specific behavior. This hybrid approach would allow Horizon+ to make conservative but sound generalizations about aliasing behavior, improving its robustness and applicability to real-world optimization pipelines.

7.2.3 Function-Level Pointer Disambiguation

The runtime disambiguation techniques introduced in this dissertation have primarily targeted loop-local transformations, effectively enabling optimizations such as loop vectorization that significantly improve performance in critical execution paths. However, many impactful compiler optimizations, such as aggressive function inlining, and interprocedural constant propagation, operate at a broader interprocedural scope and are currently hindered by conservative alias assumptions across function boundaries. Addressing these limitations requires alias information that spans functions, enabling the compiler to reason more precisely across call chains.

To this end, our pointer disambiguation can be extended to support function-level disambiguation using strategies inspired by prior work, such as [23]. Their method clones functions under the assumption of non-aliasing arguments and uses runtime checks to select between the optimized and original versions depending on the aliasing behavior at the call site. By integrating this idea with our approaches, alias resolution can be made interprocedural, allowing compilers to apply more aggressive optimizations beyond loop scopes. This extension would significantly broaden the coverage of our approaches while preserving both precision and performance.

References

[1] Rishi Surendran, Rajkishore Barik, Jisheng Zhao, and Vivek Sarkar. Inter-iteration scalar replacement using array ssa form. In Albert Cohen, editor, *Compiler Construction*, pages 40–60, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. doi:10.1007/978-3-642-54807-9_3.

[2] Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on ssa form. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, page 273–286, New York, NY, USA, 1997. Association for Computing Machinery. doi:10.1145/258915.258940.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Compilers: Principles, techniques, and tools. In *Addison-Wesley series in computer science / World student series edition*, 1986.

[4] Ralf Karrenberg and Sebastian Hack. Whole-function vectorization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, page 141–150, USA, 2011. IEEE Computer Society. doi:10.1109/CGO.2011.5764682.

[5] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 32–41, New York, NY, USA, 1996. Association for Computing Machinery. doi:10.1145/237721.237727.

[6] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848–894, jul 1999. doi:10.1145/325478.325519.

[7] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 278–289, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1250734.1250766.

[8] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, page 235–248, New York, NY, USA, 1992. Association for Computing Machinery. doi:10.1145/143095.143137.

[9] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, page 232–245, New York, NY, USA, 1993. Association for Computing Machinery. doi:10.1145/158511.158639.

[10] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, page 242–256, New York, NY, USA, 1994. Association for Computing Machinery. doi:10.1145/178243.178264.

[11] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, page 131–144, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/996841.996859.

[12] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, page

1–12, New York, NY, USA, 1995. Association for Computing Machinery. doi:10.1145/207110.207111.

[13] Jianwen Zhu. Towards scalable flow and context sensitive pointer analysis. In *Proceedings. 42nd Design Automation Conference, 2005.*, pages 831–836. IEEE, 2005. doi:10.1109/DAC.2005.193930.

[14] Yulei Sui and Jingling Xue. Svf: Interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 265–266, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2892208.2892235.

[15] Jiayi Wang, Yu Wang, Ke Wang, and Linzhang Wang. Silva: A scalable incremental layered sparse value-flow analysis. *ACM Trans. Softw. Eng. Methodol.*, mar 2025. Just Accepted. doi:10.1145/3725214.

[16] David Kozak, Codrut Stancu, Tomáš Vojnar, and Christian Wimmer. Skipflow: Improving the precision of points-to analysis using primitive values and predicate edges. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, CGO '25, page 347–361, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3696443.3708932.

[17] Tapti Palit and Pedro Fonseca. Kaleidoscope: Precise invariant-guided pointer analysis. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 561–576, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3620666.3651340.

[18] Yulei Sui, Sen Ye, Jingling Xue, and Jie Zhang. Making context-sensitive inclusion-based pointer analysis practical for compilers using parameterised summarisation. *Softw. Pract. Exper.*, 44(12):1485–1510, December 2014. doi:10.1002/spe.2214.

[19] Rupesh Nasre, Kaushik Rajan, R. Govindarajan, and Uday Khedker. Scalable context-sensitive points-to analysis using multi-dimensional bloom filters. In *Programming Languages and Systems*, pages 47–62, Berlin, Heidelberg, 12 2009. Springer Berlin Heidelberg. doi:10.1007/978-3-642

-10672-9_6.

[20] Erik Nystrom, Hong-Seok Kim, and Wen-mei Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Static Analysis*, pages 165–180, Berlin, Heidelberg, 11 2004. Springer Berlin Heidelberg. doi:10.1007/978-3-540-27864-1_14.

[21] R. Chowdhury, Peter Djeu, B. Cahoon, J. Burrill, and K. McKinley. The limits of alias analysis for scalar optimizations. In *CC*, 2004.

[22] Ankush Phulia, Vaibhav Bhagee, and Sorav Bansal. Ooelala: Order-of-evaluation based alias analysis for compiler optimization. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 839–853, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3385412.3385962.

[23] Victor Hugo Sperle Campos, Péricles Rafael Alves, Henrique Nazaré Santos, and Fernando Magno Quintão Pereira. Restrictification of function arguments. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 163–173, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2892208.2892225.

[24] Péricles Alves, Fabian Gruber, Johannes Doerfert, Alexandros Lamprineas, Tobias Grosser, Fabrice Rastello, and Fernando Magno Quintão Pereira. Runtime pointer disambiguation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, page 589–606, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2814270.2814285.

[25] Diogo N. Sampaio, Louis-Noël Pouchet, and Fabrice Rastello. Simplification and runtime resolution of data dependence constraints for loop transformations. In *Proceedings of the International Conference on Supercomputing*, ICS '17, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3079079.3079098.

[26] Dorit Naishlos. Autovectorization in gcc. In *Proceedings of the 2004 GCC*

Developers Summit, pages 105–118, 2004.

[27] Runtime checks of pointers. https://llvm.org/docs/Vectorizers.html#runtime-checks-of-pointers, 2025 (accessed Oct 13, 2025).

[28] Lars Ole Andersen and Peter Lee. Program analysis and specialization for the c programming language. 2005.

[29] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Found. Trends Program. Lang.*, 2(1):1–69, April 2015. doi:10.1561/2500000014.

[30] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, page 106–117, New York, NY, USA, 1998. Association for Computing Machinery. doi:10.1145/277650.277670.

[31] Existing alias analysis implementations and clients. https://llvm.org/docs/AliasAnalysis.html#existing-alias-analysis-implementations-and-clients, 2026 (accessed Feb 16, 2026).

[32] Cliff Click. Global code motion/global value numbering. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, page 246–257, New York, NY, USA, 1995. Association for Computing Machinery. doi:10.1145/207110.207154.

[33] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, page 145–156, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/349299.349320.

[34] The slp vectorizer. https://llvm.org/docs/Vectorizers.html#the-slp-vectorizer, 2026 (accessed Jan 9, 2026).

[35] K. D. Cooper and K. Kennedy. Fast interprocedual alias analysis. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages, POPL '89, page 49–59, New York, NY, USA, 1989. Association for Computing Machinery. doi:10.1145/75277.75282.

[36] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, page 1–14, New York, NY, USA, 1997. Association for Computing Machinery. doi:10.1145/263699.263703.

[37] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Online cycle detection and difference propagation: Applications to pointer analysis. *Software Quality Journal*, 12(4):311–337, December 2004. doi:10.1023/B:SQJO.0000039791.93071.a2.

[38] Fernando Magno Quintao Pereira and Daniel Berlin. Wave propagation and deep propagation for pointer analysis. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, page 126–135, USA, 2009. IEEE Computer Society. doi:10.1109/CGO.2009.9.

[39] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 290–299, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1250734.1250767.

[40] Ben Hardekopf and Calvin Lin. Exploiting pointer and location equivalence to optimize pointer analysis. In *Proceedings of the 14th International Conference on Static Analysis*, SAS'07, page 265–280, Berlin, Heidelberg, 2007. Springer-Verlag. doi:10.1007/978-3-540-74061-2_17.

[41] Michael G. Burke, Paul R. Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '94, page 234–250, Berlin, Heidelberg,

1994. Springer-Verlag. doi:10.1007/BFb0025882.

[42] S. Jaiswal, Uday P. Khedker, and Supratik Chakraborty. Demand-driven alias analysis : Formalizing bidirectional analyses for soundness and precision. *ArXiv*, abs/1802.00932, 2018. doi:10.48550/arXiv.1802.00932.

[43] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using cla: a million lines of c code in a second. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, page 254–263, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/378795.378855.

[44] Xin Zheng and Radu Rugina. Demand-driven alias analysis for c. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, page 197–208, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1328438.1328464.

[45] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, page 35–46, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/349299.349309.

[46] Christian Wimmer, Codrut Stancu, David Kozak, and Thomas Würthinger. Scaling type-based points-to analysis with saturation. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024. doi:10.1145/3656417.

[47] George Balatsouras and Yannis Smaragdakis. Structure-sensitive points-to analysis for c and c++. In *Sensors Applications Symposium*, pages 84–104, 09 2016. doi:10.1007/978-3-662-53413-7_5.

[48] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, page 17–30, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1926385.1926390.

[49] Michael Hind and Anthony Pioli. Which pointer analysis should i use? In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, page 113–123, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/347324.348916.

[50] Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, page 226–238, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1480881.1480911.

[51] Sen Ye, Yulei Sui, and Jingling Xue. Region-based selective flow-sensitive pointer analysis. In *International Static Analysis Symposium*, pages 319–336. Springer, Springer International Publishing, 2014. doi:10.1007/978-3-319-10936-7_20.

[52] Teck Bok Tok, Samuel Z. Guyer, and Calvin Lin. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In *Proceedings of the 15th International Conference on Compiler Construction*, CC'06, page 17–31, Berlin, Heidelberg, 2006. Springer-Verlag. doi:10.1007/11688839_3.

[53] Lian Li, Cristina Cifuentes, and Nathan Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 343–353, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2025113.2025160.

[54] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, page 289–298, USA, 2011. IEEE Computer Society. doi:10.1109/CGO.2011.5764696.

[55] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing

Zhang. Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, page 218–229, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1772954.1772985.

[56] Vaivaswatha Nagaraj and R. Govindarajan. Approximating flow-sensitive pointer analysis using frequent itemset mining. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, page 225–234, USA, 2015. IEEE Computer Society. doi:10.1109/CGO.2015.7054202.

[57] Mohamad Barbar, Yulei Sui, and Shiping Chen. Object versioning for flow-sensitive pointer analysis. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '21, page 222–235. IEEE Press, 2021. doi:10.1109/CGO51591.2021.9370334.

[58] Jisheng Zhao, Michael G. Burke, and Vivek Sarkar. Parallel sparse flow-sensitive points-to analysis. In *Proceedings of the 27th International Conference on Compiler Construction*, CC '18, page 59–70, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3178372.3179517.

[59] Swati Jaiswal, Uday P. Khedker, and Supratik Chakraborty. Bidirectionality in flow-sensitive demand-driven analysis. *Science of Computer Programming*, 190:102391, 2020. doi:10.1016/j.scico.2020.102391.

[60] ERIK MATTHEW NYSTROM. Fulcra pointer analysis framework. University of Illinois, 2012. Unpublished, Available at: .

[61] Yulei Sui, Peng Di, and Jingling Xue. Sparse flow-sensitive pointer analysis for multithreaded programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, page 160–170, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2854038.2854043.

[62] Michael Hind and Anthony Pioli. Evaluating the effectiveness of pointer

alias analyses. *Sci. Comput. Program.*, 39(1):31–55, January 2001. doi:10.1016/S0167-6423(00)00014-9.

[63] Ondřej Lhoták and Laurie Hendren. Context-sensitive points-to analysis: is it worth it? In *Proceedings of the 15th International Conference on Compiler Construction*, CC'06, page 47–64, Berlin, Heidelberg, 2006. Springer-Verlag. doi:10.1007/11688839_5.

[64] Manuvir Das, Ben Liblit, Manuel Fähndrich, and Jakob Rehof. Estimating the impact of scalable pointer analysis on optimization. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, page 260–278, Berlin, Heidelberg, 2001. Springer-Verlag. doi:10.1007/3-540-47764-0_15.

[65] Marc Shapiro and Susan Horwitz. The effects of the precision of pointer analysis. In *Proceedings of the 4th International Symposium on Static Analysis*, SAS '97, page 16–34, Berlin, Heidelberg, 1997. Springer-Verlag.

[66] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, page 54–61, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/379605.379665.

[67] Jan Hueckelheim and Johannes Doerfert. Oraql — optimistic responses to alias queries in llvm. In *Proceedings of the 52nd International Conference on Parallel Processing*, ICPP '23, page 655–664, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3605573.3605644.

[68] Michel Weber, Theodoros Theodoridis, and Zhendong Su. Relaxing alias analysis: Exploring the unexplored space. *Proc. ACM Program. Lang.*, 9(PLDI), June 2025. doi:10.1145/3729254.

[69] Markus Mock, Manuvir Das, Craig Chambers, and Susan J. Eggers. Dynamic points-to sets: A comparison with static analyses and potential applications in program understanding and optimization. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Soft-*

ware Tools and Engineering, PASTE '01, page 66–72, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/379605.379671.

[70] Jingyue Wu, Gang Hu, Yang Tang, and Junfeng Yang. Effective dynamic detection of alias analysis errors. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 279–289, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2491411.2491439.

[71] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 101–113, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1375581.1375595.

[72] Paul Feautrier. Some efficient solutions to the affine scheduling problem. i. one-dimensional time. *International journal of parallel programming*, 21(5):313–347, 1992. doi:10.1007/BF01407835.

[73] Uday Kumar Bondhugula. *Effective automatic parallelization and locality optimization using the polyhedral model*. PhD thesis, The Ohio State University, 2008.

[74] Henrique Nazaré, Izabela Maffra, Willer Santos, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintão Pereira. Validation of memory accesses through symbolic analyses. *SIGPLAN Not.*, 49(10):791–809, October 2014. doi:10.1145/2714064.2660205.

[75] Vitor Paisante, Maroua Maalej, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintão Pereira. Symbolic range analysis of pointers. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, page 171–181, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2854038.2854050.

[76] Radu Rugina and Martin C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Trans. Program.*

Lang. Syst., 27(2):185–235, March 2005. doi:10.1145/1057387.1057388.

[77] Robert A Van Engelen. Efficient symbolic analysis for optimizing compilers. In *International Conference on Compiler Construction*, pages 118–132. Springer, 2001.

[78] R Van Engelen. Symbolic evaluation of chains of recurrences for loop optimization. Technical report, Citeseer, 2000.

[79] Jeff Da Silva and J. Gregory Steffan. A probabilistic pointer analysis for speculative optimizations. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, page 416–425, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1168857.1168908.

[80] A. S. Huang, G. Slavenburg, and J. P. Shen. Speculative disambiguation: A compilation technique for dynamic memory disambiguation. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, ISCA '94, page 200–210, Washington, DC, USA, 1994. IEEE Computer Society Press. doi:10.1145/192007.192012.

[81] Manel Fernández and Roger Espasa. Speculative alias analysis for executable code. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, page 222–231, USA, 2002. IEEE Computer Society. doi:10.1109/PACT.2002.1106020.

[82] Tong Chen, Jin Lin, Xiaoru Dai, Wei-Chung Hsu, and Pen-Chung Yew. Data dependence profiling for speculative optimizations. In *International Conference on Compiler Construction*, pages 57–72. Springer, 2004. doi:10.1007/978-3-540-24723-4_5.

[83] Wonsun Ahn, Yuelu Duan, and Josep Torrellas. Dealiaser: Alias speculation using atomic region support. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 167–180, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2451116.2451136.

[84] Jin Lin, Tong Chen, Wei-Chung Hsu, Pen-Chung Yew, Roy Dz-Ching Ju, Tin-Fook Ngai, and Sun Chan. A compiler framework for speculative analysis and optimizations. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, page 289–299, New York, NY, USA, 2003. Association for Computing Machinery. doi:10.1145/781131.781164.

[85] Gregory J Chaitin, Marc A Auslander, Ashok K Chandra, John Cocke, Martin E Hopkins, and Peter W Markstein. Register allocation via coloring. *Computer languages*, 6(1):47–57, 1981. doi:10.1016/0096-0551(81)90048-5.

[86] Gregory J Chaitin. Register allocation & spilling via graph coloring. *SIGPLAN Not.*, 17(6):98–101, jun 1982. doi:10.1145/872726.806984.

[87] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, sep 1999. doi:10.1145/330249.330250.

[88] Omri Traub, Glenn Holloway, and Michael D. Smith. Quality and speed in linear-scan register allocation. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, page 142–151, New York, NY, USA, 1998. Association for Computing Machinery. doi:10.1145/277650.277714.

[89] Christian Wimmer and Michael Franz. Linear scan register allocation on ssa form. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, page 170–179, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1772954.1772979.

[90] Vivek Sarkar and Rajkishore Barik. Extended linear scan: An alternate foundation for global register allocation. In *Proceedings of the 16th International Conference on Compiler Construction*, CC'07, page 141–155, Berlin, Heidelberg, 2007. Springer-Verlag. doi:10.1007/978-3-540-71229-9_10.

[91] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof:

A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, page 120–126, New York, NY, USA, 1982. Association for Computing Machinery. doi:10.1145/800230.806987.

[92] Giuseppe Antonio Di Luna, Davide Italiano, Luca Massarelli, Sebastian Österlund, Cristiano Giuffrida, and Leonardo Querzoni. Who's debugging the debuggers? exposing debug information bugs in optimized binaries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 1034–1045, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3445814.3446695.

[93] 'noalias' and 'alias.scope' metadata. https://llvm.org/docs/LangRef.html#noalias-and-alias-scope-metadata, 2025 (accessed Oct 13, 2025).

[94] Polybench benchmark suite. https://sourceforge.net/projects/polybench/, 2016 (accessed Oct 13, 2025).

[95] Cpu spec 2017 benchmark suite. https://www.spec.org/cpu2017/Docs/overview.html#benchmarks, 2024 (accessed Oct 13, 2025).

[96] Splash 2 benchmark suite. http://web.archive.org/web/20070704172333/http://www-flash.stanford.edu/apps/SPLASH/splash2.tar.gz, 2007 (accessed Oct 13, 2025).

[97] A memo on exploration of splash-2 input sets. 2011.

[98] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, page 24–36, New York, NY, USA, 1995. Association for Computing Machinery. doi:10.1145/223982.223990.

[99] Intel 64 and ia32 architectures performance monitoring events. https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia32-architectures-performance-mon

itoring-events.html, 2017 (accessed Oct 13, 2025).

[100] Rick Kufrin. Perfsuite: An accessible, open source performance analysis environment for linux. In *In Proc. of the Linux Cluster Conference, Chapel*, 2005.

[101] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Commun. ACM*, 58(2):44–46, January 2015. doi:10.1145/2644805.

[102] Johannes Doerfert, Tobias Grosser, and Sebastian Hack. Optimistic loop optimization. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, page 292–304. IEEE Press, 2017. doi:10.1109/CGO.2017.7863748.

[103] Kenneth C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, October 1965. doi:10.1145/365628.365655.

[104] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, page 51–66, USA, 2009. USENIX Association.

[105] Intel 64 and ia-32 architectures software developer's manual volume 2b. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf, 2016 (accessed Oct 13, 2025).

[106] Intrinsic functions. https://llvm.org/docs/LangRef.html#intrinsic-functions, 2025 (accessed Oct 13, 2025).

[107] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 41–42, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3185768.3185771.

[108] Restrict keyword in llvm. https://lists.llvm.org/pipermail/llvm-dev/2019-March/131127.html, 2019 (accessed Oct 13, 2025).

[109] Daan Leijen, Benjamin G. Zorn, and Leonardo Mendonça de Moura. Mimalloc: Free list sharding in action. In *APLAS*, 2019.

[110] Mimalloc source code. https://github.com/microsoft/mimalloc, 2025 (accessed Oct 13, 2025).

[111] Khushboo Chitre, Piyus Kedia, and Rahul Purandare. The road not taken: Exploring alias analysis based optimizations missed by the compiler. *Proc. ACM Program. Lang.*, 6(OOPSLA2), oct 2022. doi:10.1145/3563316.