MACROSCOPIC DATA STRUCTURE ANALYSIS AND OPTIMIZATION

BY

CHRIS LATTNER

B.S., University of Portland, 2000
M.S., University of Illinois at Urbana-Champaign, 2002

DISSERTATION

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

Urbana, Illinois

Abstract

Providing high performance for pointer-intensive programs on modern architectures is an increasingly difficult problem for compilers. Pointer-intensive programs are often bound by memory latency and cache performance, but traditional approaches to these problems usually fail: Pointer-intensive programs are often highly-irregular and the compiler has little control over the layout of heap allocated objects.

This thesis presents a new class of techniques named "Macroscopic Data Structure Analyses and Optimizations", which is a new approach to the problem of analyzing and optimizing pointer-intensive programs. Instead of analyzing individual load/store operations or structure definitions, this approach identifies, analyzes, and transforms entire memory structures as a unit. The foundation of the approach is an analysis named Data Structure Analysis and a transformation named Automatic Pool Allocation. Data Structure Analysis is a context-sensitive pointer analysis which identifies data structures on the heap and their important properties (such as type safety). Automatic Pool Allocation uses the results of Data Structure Analysis to segregate dynamically allocated objects on the heap, giving control over the layout of the data structure in memory to the compiler.

Based on these two foundation techniques, this thesis describes several performance improving optimizations for pointer-intensive programs. First, Automatic Pool Allocation itself provides important locality improvements for the program. Once the program is pool allocated, several pool-specific optimizations can be performed to reduce inter-object padding and pool overhead. Second, we describe an aggressive technique, Automatic Pointer Compression, which reduces the size of pointers on 64-bit targets to 32-bits or less, increasing effective cache capacity and memory bandwidth for pointer-intensive programs.

This thesis describes the approach, analysis, and transformation of programs with macroscopic techniques, and evaluates the net performance impact of the transformations. Finally, it describes

a large class of potential applications for the work in fields such as heap safety and reliability, program understanding, distributed computing, and static garbage collection.

To Tanya, for her unwavering love and support.

Acknowledgments

This thesis would not be possible without the support of many people who have helped me in ways both large and small.

In particular, I would like to thank my advisor, Vikram Adve, for his support, patience, and especially his trust and respect. He taught me how to communicate ideas more effectively, gave me the freedom to investigate the (broad and sometimes strange) areas that interest me, and contributed many ideas to this work. Few other advisors would allow a motivated student to go off and build an entire compiler to support their research.

My wife Tanya is an unshakable source of support, understanding, and love. She helped me make it through the occasionally grueling all nighters and other challenging parts of these last five years, selflessly supporting me even when under pressures from her own research work and job. In addition to support of my research, she continues to enrich my life as a whole.

I have deeply enjoyed my interactions and friendships with the members of the LLVM research group as well as the open-source community we have built around LLVM. Both have provided important insights, hard problems, and a desire to make LLVM as stable, robust, and extensible as possible. Special thanks go to Misha Brukman for reading (and rereading) many of my papers with his particularly critical eye for incorrect-hyphenation and mispellings.

I would like to thank the UIUC Classical Fencing Club as a whole, and John Mainzer and Luda Yafremava in particular, for absorbing many of the frustrations and craziness accumulated over the course of this work. They provided an important outlet and taught me physical awareness, flexibility, and dexterity that I did not think was possible. They are also responsible for keeping random workplace violence to an tolerable level, for which my colleagues are undoubtedly thankful!

Finally, I would like to thank Steven Vegdahl, who encouraged me to pursue graduate studies and whose infectious love of compilers started me on this path in the first place.

Table of Contents

List of Figures

Chapter 1

Introduction

Memory system performance is one of the main factors limiting application performance on modern architectures. Today, modern compilers are able to aggressively analyze and optimize programs that use dense arrays, sometimes providing multiple factors of performance improvement. Pointer-based recursive data structures, on the other hand, have proven to be much more difficult – both to analyze and optimize – and thus compilers have had much less success improving the performance of these programs.

The primary existing approaches for analyzing and transforming pointer-intensive programs fall into two broad categories: scalar optimizations (such as register promotion [37], dead store elimination, greedy prefetching [94], etc.), and structure transformations (structure field reordering, structure splitting [28], jump-pointer prefetching [112, 94], etc.). These approaches are attractive because they only require local changes to the program: a few instructions in the first case and a single structure definition in the second.

Unfortunately, this is precisely the property that limits the potential gain of these approaches. Scalar optimizations are inherently limited to making local performance improvements because they only modify a few instructions at a time (for example, a particular load or store). The second approach is more aggressive and more promising, but generally requires the program to be written in a type-safe source-language and is limited to transforming *all instances* of a particular type or none of them (for example, a field that is unused in one instance of a data structure, but not another, cannot be removed from either).

Most importantly, neither of these approaches is able to attack the root cause of the problem: the compiler cannot analyze or control the layout of objects on the heap. In particular, the reason that

recursive data structures exhibit poor locality is that their nodes are often distributed throughout the heap with little correlation between the layout of the nodes and the access/traversal pattern of the program. Because the access patterns of these recursive data structures are not directly connected to the layout of the objects on the heap, standard techniques for improving the cache performance of dense arrays cannot be applied to nodes in a recursive data structure.

Aggressively optimizing programs that heavily use recursive data structures is inherently difficult for several reasons. First, interprocedural analysis is *required* for any real-world program: recursive data structures are often created, traversed, and destroyed with recursive functions, are often passed throughout the program, and often used to build larger aggregate structures (e.g. a list of lists). Second, *extremely aggressive* forms of interprocedural analysis are required: modern software design techniques encourage the use of modular and reusable data structure libraries, and these libraries may be used in different ways in different portions of the program. Ideally, we would like to be able to optimize individual instances of a particular data structure, even if all of the instances of that type are processed and created with common functions (traditional scalable points-to analyses are insufficient for these programs). Third, compilers for statically compiled languages generally do not have control over the memory management runtime, greatly limiting the information and control it has over the runtime layout of a data structure. Finally, compilers designed to optimize unsafe languages (like C or C++) must correctly handle programs that cast pointers or rely on the precise layout of data in memory (e.g., programs that copy structures to disk or across a network).

Throughout this work, we use the term "data structure" to mean an *instance* of a heap allocated recursive data structure potentially formed with multiple node types (e.g. a graph with 'edge' and 'node' objects). This work is not concerned with classification of a data structure instance as some high-level conceptual type (e.g. a binary tree or a linked list), instead, we focus on the properties that are independent of the high level conceptual type (e.g. node layout properties).

Prior to our work, shape analysis was the only extant approach for performing macroscopic analyses of programs that use data structures. Shape analysis is able to provide strong classification of data structures in the program as various high-level types, such as a singly- or doubly-linked list, a binary tree, etc. Unfortunately, shape analyses cannot handle non-type-safe programs,

provide no mechanism for controlling the layout of heap-based data structures, and are extremely expensive [77, 144, 65].

This thesis describes a **macroscopic approach for analyzing and transforming heap-allocated data structures** which addresses the deficiencies of previous approaches by using aggressive (but practical) static analysis and transformation techniques to identify and control recursive data structures. Macroscopic techniques aim to:

- ... analyze and transform an *entire* data structure as a unit and each distinct *instance* of a data structure independently.
- ... give partial control over the heap-layout of data structure instances to the compiler, allowing it reason about and optimize some important layout properties.
- ... tune individual instances of data structures to the clients that use them. Reusable data structure libraries are very common, but different clients have different usage behaviors, and each application may contain many distinct clients of the library.
- ... provide a framework for existing approaches that use mod/ref or alias analysis, and support techniques that are traditionally implemented by changing structure type layout for entire programs (e.g., structure reordering/fission [28], instance interleaving [136], jump-pointer prefetching [112], etc), sometimes making them more powerful in the process.
- ... be suitable for inclusion in a commercial compiler. Our implementation of these techniques are scalable to large programs, work with incomplete programs, are safe in the presence of exception handling and `setjmp`/`longjmp` calls, correctly determine whether a data structure is accessed in a type-safe way, etc.

By achieving these goals, we show that macroscopic techniques can dramatically improve the performance of heap intensive programs, with purely automatic techniques which are applied at program link-time. This work is implemented in the context of the LLVM Compiler Infrastructure, which was built to support the aggressive link-time analysis and optimization required by this work. LLVM is described in Chapter 2 to provide the context for this work, portions of which were published in [87, 3, 88].

1.1 Foundations of the Macroscopic Approach

Our implementation of macroscopic algorithms are built on a foundation consisting of two techniques: Data Structure Analysis and Automatic Pool Allocation. Based on this analysis and transformation, many new techniques are feasible.

1.1.1 Data Structure Analysis

Data Structure Analysis (DSA) is a context- and field-sensitive pointer analysis. It is used to identify the connectivity of memory objects in a program, identify instances of data structures, and capture important properties of these structures (such as whether accesses to the objects are type-safe).

DSA is an aggressive interprocedural analysis which uses full acyclic call paths to name heap and stack objects (it is "fully" context sensitive), allowing it to identify disjoint instances of data structures, even if they are created and processed by common helper functions. DSA can support a superset of the clients supported by most flow-insensitive interprocedural alias, mod-ref, and call graph analyses, in addition to supporting the macroscopic analyses described throughout this thesis. DSA supports the full generality of C/C++ programs and provides conservatively correct analysis of incomplete programs and libraries.

The primary research contributions of DSA are:

(i) New techniques used to achieve its speed, scalability, and low memory footprint when analyzing large programs, despite its aggressive analysis. We show that DSA uses little memory and is fast and scalable in our experiments on programs spanning 4-5 orders of magnitude of code size (past 200,000 lines of code), never taking more than 3.2s on these codes. We describe why we believe that it will continue to scale well to larger programs in Sections 3.2.5 and 3.4.2. DSA is the first fully context sensitive algorithm we are aware of that analyzes programs in a fraction of the time taken to compile the program with a standard optimizing compiler (GCC). Further, the fraction of compile time used by DSA is quite small: always less than 6% in our experiments.

(ii) Use of a novel extension to Tarjan's Strongly Connected Component (SCC) finding algorithm

that allows incremental discovery of SCCs in the call graph, even when edges are dynamically discovered and added to the call graph.

(iii) Uses of a simple mechanism (fine-grain incompleteness tracking) to solve several hard problems in pointer analysis, including the use of speculative type information, dynamic discovery of the call graph without iteration, and conservatively correct handling of incomplete programs.

DSA is used by all macroscopic techniques and is described in detail in Chapter 3. Because all of the work in this thesis depends on DSA, we pay special attention to making sure that DSA is suitable for use in a commercial compiler, which includes being fast enough for plausible use, fully supporting incomplete programs, and supporting the full generality of C (setjmp/longjmp, function pointers, and non-type-safe pointer casts). We evaluate the precision of DSA when used as a standard pointer analysis in Chapter 4.

1.1.2 Automatic Pool Allocation

Automatic Pool Allocation transforms the data structures identified by DSA to segregate the memory for each data structure into a "pool" or "region" of memory. For example, if DSA identifies two disjoint linked-lists as part of its execution, Automatic Pool Allocation will transform the program to create one region of memory for each list, and use that region to manage all of the memory allocated for each list. Automatic Pool Allocation ensures that the dynamic lifetime of the pool to be a superset of the dynamic lifetime of the data structure being pool allocated.

The primary motivation of the pool allocation transformation is to give *partial control of the dynamic layout of a data structure* to the compiler. While prior compiler transformations have provided limited control over layout of heap objects (garbage collector or allocation library heuristics [68, 64, 12, 39, 30, 119, 29], for example), none have been able to control the layout of a data structure at the granularity of individual instances of the data structure, and none have been able to support subsequent aggressive compiler transformations that optimize data structures on a per-instance basis (e.g., the simple ones in Chapter 6 or the aggressive one in Chapter 7). In particular, because all of the nodes of transformed data structures are managed by a compiler-controlled runtime library, compiler transformations can emit code that identifies and manipulates all of the

allocated objects in the data structure at program runtime, which enables novel transformations (like the Pointer Compression transformation described below).

The primary research contributions of Automatic Pool Allocation are:

(i) We show that the algorithm succeeds in segregating recursive data structures on the heap, providing a substantial performance improvement for several programs. We experimentally find that the algorithm improves the performance of several programs by 10-20%, speeds up two by about 2x and two others by about 10x, and explain the source of these improvements.

(ii) Unlike previous related approaches [134, 133, 4, 66, 31, 26], *all* of which require a type-safe input program, Automatic Pool Allocation supports the full generality of C and C++ programs (including indirect function calls, mutually recursive functions, variable argument functions, lack of type-safety, setjmp/longjmp, etc.).

(iii) Our algorithm is the first to perform region inference based on a scalable pointer analysis (DSA), which allows us to partition heap data by *reachability*. Work concurrent to ours [26] uses a somewhat similar approach, but uses a non-scalable analysis, does not handle global variables at all, requires type-safety, and has other limitations compared to our approach (as described in Section 5.6).

(iv) We present a simple strategy for correctly handling indirect function calls in arbitrary C programs without making the core transformation more complex.

(v) The algorithm computes static mapping information from pointers to the pools that they point into. We are the first to demonstrate that region inference and this mapping information can be used to support aggressive follow-on techniques like Transparent Pointer Compression.

In addition to the research contributions, we show that the analysis and transformation required to perform this optimization both require very little compile time or memory (less than 1.3s (including DSA time) for the programs we tested, which include codes up to 100,000 lines of code). To put this in perspective, this is at most 3% of the time required for GCC to compile the program (on the programs we tried) at its -O3 level of optimization. We feel that the amount of

resources used is quite reasonable for an aggressive optimizing transformation targetting memory system performance.

The Automatic Pool Allocation algorithm is described in detail in Chapter 5. Portions of this work were published in [89].

1.2 Applications of Macroscopic Techniques

Building on the foundation of DSA and Automatic Pool Allocation, a wide range of new macroscopic techniques are possible. This thesis explores several macroscopic techniques which target improved performance, described briefly below.

1.2.1 Simple Pool Allocation Optimizations

The first and most straight-forward application is a collection of simple improvements to the pool allocated code. Because the pool allocator has complete control over the pool runtime library, we can expose a richer interface to the compiler than what is provided by the standard C library `malloc` and `free` family of functions. In particular, if the compiler can prove that a pool of memory only contain nodes that require 4-byte alignment, it can lower the alignment requirement for the pool (which defaults to 8-byte alignment), potentially reducing inter-object padding. Likewise, if the compiler can prove the memory is never deallocated from a pool, it can inform the runtime that it does not need to keep track of any metadata for objects in the pool (reducing allocation time and eliminating a per-object header word).

The key contribution that pool allocation provides is by *partitioning distinct data structures in the heap*, so that these decisions can be made on a per-data-structure basis. For example, this allows some data structures in the program to be fully aligned where necessary, and others to use less alignment when possible. Without pool allocation, even with a mutable runtime library, these sorts of decisions would have to be made on global (per-program) basis, which would rarely allow any improvement. Chapter 6 describes and evaluates these techniques in more detail, showing that simple optimizations like this can provide up to a 40% performance improvement over that already provided by pool allocation alone.

1.2.2 Transparent Pointer Compression

64-bit systems are becoming both increasingly important and increasingly common. Unfortunately, use of 64-bit pointers can dramatically impact the performance of pointer-intensive programs, as they require twice as much memory, cache space, and memory bandwidth to process as 32-bit pointers. Transparent Pointer Compression offers two ways of combating this problem: static and dynamic pointer compression.

Static Pointer Compression automatically identifies and transforms instances of type-safe data structures, replacing pointers in the data structure with smaller integer offsets from the start of the pool they are located in. Because pool allocation divides a program up into pools, it allows recursive data structures to each grow to 2^{32} bytes (and in some cases 2^{32} nodes), without encountering a runtime error. However, the possibility of this runtime error is not acceptable for all domains.

Dynamic Pointer Compression solves this problem by speculatively compressing 64-bit pointers to 32-bit indices in type-safe data structures, while allowing them to grow back to full 64-bit indexes when needed (rewriting memory as needed). This allows the compiler to speculate that the data structures will be small without losing generality to programs with large data sizes.

Chapter 7 shows that Static Pointer Compression can speed up pointer intensive programs from 20% to 2x in extreme cases (over pool allocation), matching the performance of programs compiled to use native 32-bit pointers in many cases. In cases where use of 64-bit mode enables features that are not available in 32-bit mode (e.g. the AMD64 architecture), pointer compression can even beat native 32-bit performance.

Pointer Compression is due to be published in [90].

1.2.3 Other Macroscopic Techniques

Macroscopic techniques can also be used for a wide range of other non-performance related purposes. In particular, they may be used to improve program checkpointing (only checkpointing data structures that have changed since the last checkpoint), partitioning memory for embedded systems and non-traditional processor architectures, connectivity-based garbage collection [74], program understanding, program visualization, and even data marshaling for remote procedure calls (passing pointer based data structures by converting pointers into indexes). Finally, our group is investi-

gating use of macroscopic techniques to provide program heap safety for languages with explicit deallocation [49], and to guarantee that the points-to and call graph computed for a program is sound. We discuss some of these techniques in more detail in Chapter 8.

1.3 Research Contributions of this Thesis

The high-level contribution of this work is a **new approach for analysis and transformation of heap-intensive programs written in arbitrary source-languages**. This approach has a broad range of potential applications, some of which we explore in detail, others we only discuss. More specifically, the high-level contributions of this thesis are (see the individual chapters for more detail):

- Data Structure Analysis: An aggressive and scalable context- and field-sensitive heap analysis that safely supports the full generality of C programs (lack of type-safety, variable argument functions, `setjmp/longjmp`, incomplete programs, etc). In addition, DSA is at least an order of magnitude faster than previous fully context-sensitive algorithms [92, 140, 103], making it the first to take a small fraction of the time required to compile the program with a standard optimizing compiler.

- Automatic Pool Allocation: The first fully automatic compiler transformation to partition the data structures of a program on the heap while retaining enough information for subsequent compiler analysis and optimization. Pool allocation provides the compiler with information about and partial control over the layout of memory objects on a points-to graph node granularity, and can substantially improve the performance of recursive data structure intensive programs.

- Pointer Compression: The first transformation to selectively shrink pointers in selected recursive data structures from 64-bit to 32-bit, dramatically reducing the memory footprint and working set of pointer intensive programs on 64-bit systems. We describe (but have not yet implemented) a fully general version of the transformation which can speculatively shrink pointers while retaining the ability to dynamically expand them at run-time if 64-bit generality is required.

- Pool Micro-optimizations: A collection of simple optimizations which can potentially be applied to many pool-based and region-based runtime libraries, aimed at improving cache density by reducing inter-node padding (memory allocation headers, alignment padding, etc) and using the semantics of the pool library to eliminate operations.

1.4 Thesis Organization

Chapter 2 describes background information about the LLVM Compiler Infrastructure, which was developed to support this work. Following that, Chapters 3 and 5 describe the two foundations of macroscopic techniques: Data Structure Analysis and Automatic Pool Allocation. Chapter 4 evaluates the precision of DSA for alias analysis applications and Chapter 6 describes the suite of simple optimizations used to improve the performance of pool allocated programs. Chapter 7 describes Transparent Pointer Compression, an aggressive macroscopic transformation. Following this, Chapter 8 describes macroscopic applications that are not a core part of this thesis: both those explored primarily by other people and those that are still speculative. Finally, Chapter 9 concludes the work.

Chapter 2

The LLVM Compiler Infrastructure

Macroscopic data structure analysis and optimization inherently requires aggressive interprocedural analysis and transformation to be effective. When this work started, no open-source compiler system was available that provided the capabilities needed. As such, we developed and built the LLVM Compiler Infrastructure (concurrently with the work in the rest of this thesis) to support aggressive interprocedural optimization, and to build a novel program representation that reduces the difficulty of implementing these aggressive techniques.

This chapter describes some of the important details of the LLVM Compiler System (which is now used for far more than the macroscopic techniques in this thesis), including the type system implemented and instruction representation.

2.1 Introduction

Modern applications are increasing in size, change their behavior significantly during execution, support dynamic extensions and upgrades, and often have components written in multiple different languages. While some applications have small hot spots, others spread their execution time evenly throughout the application [34]. In order to maximize the efficiency of all of these programs, we believe that program analysis and transformation must be performed throughout the lifetime of a program. Such "lifelong code optimization" techniques encompass interprocedural optimizations performed at link-time (to preserve the benefits of separate compilation), machine-dependent optimizations at install time on each system, dynamic optimization at runtime, and profile-guided optimization between runs ("idle time") using profile information collected from the end-user.

Program optimization is not the only use for lifelong analysis and transformation. Other applications of static analysis are fundamentally interprocedural, and are therefore most convenient to perform at link-time (examples include *macroscopic data structure optimization*, static debugging, static leak detection [69], and many other transformations). Sophisticated analyses and transformations are being developed to enforce program safety, but must be done at software installation time or load-time [48]. Allowing lifelong reoptimization of the program gives architects the power to evolve processors and exposed interfaces in more flexible ways [27, 50], while allowing legacy applications to run *well* on new systems.

This chapter describes **LLVM** — Low-Level Virtual Machine — a compiler framework that aims to make lifelong program analysis and transformation available for arbitrary software, and in a manner that is transparent to programmers. LLVM achieves this through two parts: (a) *a code representation* with several novel features that serves as a common representation for analysis, transformation, and code distribution; and (b) *a compiler design* that exploits this representation to provide a combination of capabilities that is not available in any previous compilation approach we know of.

The LLVM code representation describes a program using an abstract RISC-like instruction set but with key higher-level information for effective analysis. This includes type information, explicit control flow graphs, and an explicit dataflow representation (using an infinite, typed register set in Static Single Assignment form [40]). There are several novel features in the LLVM code representation: (a) A low-level, *language-independent* type system that can be used to *implement* data types and operations from high-level languages, exposing their implementation behavior to all stages of optimization. This type system includes the type information used by sophisticated (but language-independent) techniques, such as algorithms for pointer analysis, dependence analysis, and data transformations. (b) Instructions for performing type conversions and low-level address arithmetic while preserving type information. (c) Two low-level exception-handling instructions for implementing language-specific exception semantics, while explicitly exposing exceptional control flow to the compiler.

The LLVM representation is *source-language-independent*, for two reasons. First, it uses a low-level instruction set and memory model that are only slightly richer than standard assembly

languages, and the type system does not *prevent* representing code with little type information. Second, it does not impose any particular runtime requirements or semantics on programs. Nevertheless, it's important to note that LLVM is *not intended to be a universal compiler IR*. In particular, LLVM does not represent high-level language features directly (so it cannot be used for some language-dependent transformations), nor does it capture machine-dependent features or code sequences used by back-end code generators (it must be lowered to do so).

Because of the differing goals and representations, *LLVM is complementary to high-level virtual machines* (e.g., SmallTalk [47], Self [137], JVM [93], Microsoft's CLI [95], and others), and *not an alternative to these systems*. It differs from these in three key ways. First, LLVM has no notion of high-level constructs such as classes, inheritance, or exception-handling semantics, even when compiling source languages with these features. Second, LLVM does not specify a runtime system or particular object model: it is low-level enough that the runtime system for a particular language can be implemented in LLVM itself. Indeed, LLVM can be used to *implement* high-level virtual machines. Third, LLVM does not guarantee type safety, memory safety, or language interoperability any more than the assembly language for a physical processor does.

The LLVM compiler framework exploits the code representation to provide a combination of five capabilities that we believe are important in order to support lifelong analysis and transformation for arbitrary programs. In general, these capabilities are quite difficult to obtain simultaneously, but the LLVM design does so inherently:

(1) *Persistent program information*: The compilation model preserves the LLVM representation throughout an application's lifetime, allowing sophisticated optimizations to be performed at all stages, including runtime and idle time between runs.

(2) *Offline code generation*: Despite the last point, it is possible to compile programs into efficient native machine code *offline*, using expensive code generation techniques not suitable for runtime code generation. This is crucial for performance-critical programs.

(3) *User-based profiling and optimization*: The LLVM framework gathers profiling information at run-time *in the field* so that it is representative of actual users, and can apply it for profile-guided transformations both at run-time and in idle time[1].

[1] An idle-time optimizer has not yet been implemented in LLVM.

(4) *Transparent runtime model*: The system does not specify any particular object model, exception semantics, or runtime environment, thus allowing any language (or combination of languages) to be compiled using it.

(5) *Uniform, whole-program compilation*: Language-independence makes it possible to optimize and compile all code comprising an application in a uniform manner (after linking), including language-specific runtime libraries and system libraries.

We believe that *no previous system provides all five of these properties.* Source-level compilers provide #2 and #4, but do not attempt to provide #1, #3 or #5. Link-time interprocedural optimizers [54, 9, 76], common in commercial compilers, provide the additional capability of #1 and #5 but only up to link-time. Profile-guided optimizers for static languages provide benefit #2 at the cost of transparency, and most crucially do not provide #3. High-level virtual machines such as JVM or CLI provide #3 and partially provide #1 and #5, but do not aim to provide #4, and either do not provide #2 at all or without #1 or #3. Binary runtime optimization systems provide #2, #4 and #5, but provide #3 only at runtime and to a limited extent, and most importantly do not provide #1. We explain these in more detail in Section 2.3.

We evaluate the effectiveness of the LLVM system with respect to three issues: (a) the size and effectiveness of the representation, including the ability to extract useful type information for C programs; (b) the compiler performance (not the performance of generated code which depends on the particular code generator or optimization sequences used); and (c) examples illustrating the key capabilities LLVM provides for several challenging compiler problems.

Our experimental results show that the LLVM compiler (using Data Structure Analysis from Chapter 3) can extract reliable type information for an average of 68% of the static memory access instructions across a range of SPECINT 2000 C benchmarks, and for virtually all the accesses in more disciplined programs. We also discuss based on our experience that the type information captured by LLVM is enough to enable aggressive transformations that would traditionally be attempted only on type-safe languages in source-level compilers through the use of macroscopic techniques. Code size measurements show that the LLVM representation is comparable in size to X86 machine code (a CISC architecture) and roughly 25% smaller than RISC code on average, despite capturing much richer type information as well as an infinite register set in SSA form.

Finally, we present example timings showing that the LLVM representation supports extremely fast interprocedural optimizations.

The primary languages supported by LLVM are C and C++, which are traditionally compiled entirely statically. LLVM also supports several other languages to various extents, and a Java front-end is currently being developed. LLVM is freely available under a non-restrictive license from the LLVM home-page (`http://llvm.cs.uiuc.edu/`), and has been used for many commercial and academic projects to date..

The rest of this chapter is organized as follows. Section 2.2 describes the LLVM code representation. Section 2.3 then describes the design of the LLVM compiler framework. Section 2.4 discusses our evaluation of the LLVM system as described above. Section 2.5 compares LLVM with related previous systems. Section 2.6 concludes with a summary of the paper.

2.2 Program Representation

The code representation used by LLVM is one of the key factors that differentiates it from other systems. The representation is designed to provide high-level information about programs that is needed to support sophisticated analyses and transformations (such as macroscopic techniques), while being low-level enough to represent arbitrary programs and to permit extensive optimization in static compilers. This section gives an overview of the LLVM instruction set and describes the language-independent type system, the memory model, exception handling mechanisms, and the offline and in-memory representations. The detailed syntax and semantics of the representation are defined in the LLVM reference manual [86].

2.2.1 Overview of the LLVM Instruction Set

The LLVM instruction set captures the key operations of ordinary processors but avoids machine-specific constraints such as physical registers, pipelines, and low-level calling conventions. LLVM provides an infinite set of typed virtual registers which can hold values of *primitive types* (Boolean, integer, floating point, and pointer). The virtual registers are in Static Single Assignment (SSA) form [40]. LLVM is a load/store architecture: programs transfer values between registers and memory solely via `load` and `store` operations using typed pointers. The LLVM memory model is

described in Section 2.2.3.

The entire LLVM instruction set consists of only 31 opcodes. This is possible because, first, we avoid multiple opcodes for the same operations[2]. Second, most opcodes in LLVM are overloaded (for example, the `add` instruction can operate on operands of any integer or floating point operand type). Most instructions, including all arithmetic and logical operations, are in three-address form: they take one or two operands and produce a single result.

LLVM uses SSA form as its primary code representation, i.e., each virtual register is written in exactly one instruction, and each use of a register is dominated by its definition. Memory locations in LLVM are *not* in SSA form because many possible locations may be modified at a single store through a pointer, making it difficult to construct a reasonably compact, explicit SSA code representation for such locations. The LLVM instruction set includes an explicit `phi` instruction, which corresponds directly to the standard (non-gated) ϕ function of SSA form. SSA form provides a compact def-use graph that simplifies many dataflow optimizations and enables fast, flow-insensitive algorithms to achieve many of the benefits of flow-sensitive algorithms without expensive dataflow analysis. Non-loop transformations in SSA form are further simplified because they do not encounter anti- or output dependences on SSA registers. Non-memory transformations are also greatly simplified because (unrelated to SSA) registers cannot have aliases.

LLVM also makes the Control Flow Graph (CFG) of every function explicit in the representation. A function is a set of basic blocks, and each basic block is a sequence of LLVM instructions, ending in exactly one terminator instruction (branches, return, `unwind`, or `invoke`; the latter two are explained later below). Each terminator explicitly specifies its successor basic blocks.

2.2.2 Language-Independent Type Information, Cast, and GetElementPtr

One of the fundamental design features of LLVM is the inclusion of a language-independent type system. Every SSA register and explicit memory object has an associated type, and all operations obey strict type rules. This type information is used in conjunction with the instruction opcode to determine the exact semantics of an instruction (e.g. floating point vs. integer add). This type information enables a broad class of *high-level* transformations on *low-level* code (for example, see

[2]For example, there are no unary operators: `not` and `neg` are implemented in terms of `xor` and `sub`, respectively.

Section 2.4.1). In addition, type mismatches are useful for detecting optimizer bugs.

The LLVM type system includes source-language-independent primitive types with predefined sizes (void, bool, signed/unsigned integers from 8 to 64 bits, and single- and double-precision floating-point types). This makes it possible to write portable code using these types, though non-portable code can be expressed directly as well. LLVM also includes (only) five derived types: pointers, arrays, structures, functions, and SIMD vectors. We believe that most high-level language data types are eventually represented using some combination of these five types in terms of their operational behavior. For example, C++ classes with inheritance are implemented using structures, functions, and arrays of function pointers, as described in Section 7.

Equally important, the five derived types above capture the type information used even by sophisticated language-independent analyses and optimizations. For example, field-sensitive points-to analyses like Data Structure Analysis, call graph construction (including for object-oriented languages like C++), scalar promotion of aggregates, and structure field reordering transformations [28], only use pointers, structures, functions, and primitive data types, while array dependence analysis and loop transformations use all those plus array types.

Because LLVM is language independent and must support weakly-typed languages, *declared* type information in a legal LLVM program may not be reliable. Instead, some pointer analysis algorithm must be used to distinguish memory accesses for which the type of the pointer target is reliably known from those for which it is not. The most aggressive algorithm currently included with LLVM is Data Structure Analysis (described in Chapter 3). Our results show that despite allowing values to be arbitrarily cast to other types, reliable type information is available for a large fraction of memory accesses in C programs compiled to LLVM.

The LLVM '`cast`' instruction is used to convert a value of one type to another arbitrary type, and is the *only* way to perform such conversions. Casts thus make all type conversions explicit, including type coercion (there are no mixed-type operations in LLVM), explicit casts for physical subtyping, and reinterpreting casts for non-type-safe code. A program without `cast`s is necessarily type-safe (in the absence of memory access errors, e.g., array overflow [48]).

A critical difficulty in preserving type information for low-level code is implementing address arithmetic. The `getelementptr` instruction is used by the LLVM system to perform pointer arith-

metic in a way that both preserves type information and has machine-independent semantics. Given a typed pointer to an object of some aggregate type, this instruction calculates the address of a sub-element of the object in a type-preserving manner (effectively a combined '.' and '[ ]' operator for LLVM).

For example, the C statement "`X[i].a = 1;`" could be translated into the pair of LLVM instructions:

```
%p = getelementptr %xty* %X, int %i, ubyte 3;
store int 1, int* %p;
```

where we assume a is field number 3 within the structure `X[i]`, and the structure is of type `%xty`. Multiple structure and array index values can be specified in one `getelementptr` instruction to index into nested aggregate types.

```
struct RT {       /* Structure with complex types */
  char A; int B[10][20]; char C;
};
struct ST {     /* ST contains an instance of RT embedded in it */
  int X;  double Y; struct RT Z;
};

int *foo(struct ST *s) {
  return &s[1].Z.B[5][13];
}
```

Figure 2.1: C code for complex memory addressing

The example in Figure 2.1 is C code fragment that defines two structure types and a function that performs complex indexing. Figure 2.2 shows the LLVM code generated by the C front-end, with commentary, illustrating the `getelementptr` instruction.

```
; LLVM type definitions
%RT = type { sbyte, [10 x [20 x int]], sbyte }
%ST = type { int, double, %RT }

; Function body...
%ST* %s = ...
%tmp = getelementptr %ST* %s, int 1, ubyte 2, ubyte 1, uint 5, uint 13
```

Figure 2.2: LLVM code for complex memory addressing

Making all address arithmetic explicit is important so that it is exposed to all LLVM optimizations (most importantly, reassociation and redundancy elimination); getelementptr achieves this

without obscuring the type information. Load and store instructions take a single pointer and do not perform any indexing, which makes the processing of memory accesses simple and uniform.

2.2.3 Explicit Memory Allocation and Unified Memory Model

LLVM provides instructions for typed memory allocation. The `malloc` instruction allocates one or more elements of a specific type on the heap, returning a typed pointer to the new memory. The `free` instruction releases memory allocated through `malloc`[3]. The `alloca` instruction is similar to `malloc` except that it allocates memory in the stack frame of the current function instead of the heap, and the memory is automatically deallocated on return from the function. All stack-resident data (including "automatic" variables) are allocated explicitly using `alloca`.

In LLVM, all addressable objects ("lvalues") are explicitly allocated. Global variable and function definitions define a symbol which provides the address of the object (not the object itself), and all stack memory is explicitly allocated with the `alloca` instruction. This gives a unified memory model in which all memory operations, including call instructions, occur through typed pointers. There are no implicit accesses to memory, simplifying memory access analysis, and the representation needs no "address of" operator.

2.2.4 Function Calls and Exception Handling

For ordinary function calls, LLVM provides a `call` instruction that takes a typed function pointer (which may be a function name or an actual pointer value) and typed actual arguments. This abstracts away the calling conventions of the underlying machine and simplifies program analysis.

One of the most unusual features of LLVM is that it provides an explicit, low-level, machine-independent mechanism to implement exception handling in high-level languages. In fact, the same mechanism also supports `setjmp` and `longjmp` operations in C, allowing these operations to be analyzed and optimized in the same way that exception features in other languages are. The common exception mechanism is based on two instructions, `invoke` and `unwind`.

The `invoke` and `unwind` instructions together support an abstract exception handling model logically based on stack unwinding (though LLVM-to-native code generators may use either "zero

[3]When native code is generated for a program, `malloc` and `free` instructions are converted to the appropriate native function calls, allowing custom memory allocators to be used.

cost" table-driven methods [22] or `setjmp/longjmp` to implement the instructions). `invoke` is used to specify exception handling code that must be executed during stack unwinding for an exception. `unwind` is used to throw an exception or to perform a `longjmp`. We first describe the mechanisms and then describe how they can be used for implementing exception handling.

The `invoke` instruction works just like a `call`, but specifies an extra basic block that indicates the starting block for an unwind handler. When the program executes an `unwind` instruction, it logically unwinds the stack until it removes an activation record created by an `invoke`. It then transfers control to the basic block specified by the `invoke`. These two instructions expose exceptional control flow in the LLVM CFG.

These two primitives can be used to implement a wide variety of exception handling mechanisms. We implemented full support for C's `setjmp/longjmp` calls and the C++ exception model; in fact, both coexist cleanly in our implementation [33]. At a call site, if some code must be executed when an exception is thrown (for example, `setjmp`, "catch" blocks, or automatic variable destructors in C++), the code uses the `invoke` instruction for the call. When an exception is thrown, this causes the stack unwinding to stop in the current function, execute the desired code, then continue execution or unwinding as appropriate.

```
{
  AClass Obj;    // Has a destructor
  func();        // Might throw; must execute destructor
  ...
}
```

Figure 2.3: C++ exception handling example

For example, consider Figure 2.3, which shows a case where "cleanup code" needs to be generated by the C++ front-end. If the '`func()`' call throws an exception, C++ guarantees that the destructor for the `Object` object will be run. To implement this, an `invoke` instruction is used to halt unwinding, the destructor is run, then unwinding is continued with the `unwind` instruction. The generated LLVM code is shown in Figure 2.4. Note that a front-end for Java would use similar code to unlock locks that are acquired through synchronized blocks or methods when exceptions are thrown.

A key feature of our approach is that the complex, language-specific details of what code

```
  ...
  ; Allocate stack space for object:
  %Obj = alloca %AClass, uint 1
  ; Construct object:
  call void %AClass::AClass(%AClass* %Obj)
  ; Call ''func()'':
  invoke void %func() to label %OkLabel
                 unwind to label %ExceptionLabel
OkLabel:
  ; ... execution continues...
ExceptionLabel:
  ; If unwind occurs, excecution continues
  ; here.  First, destroy the object:
  call void %AClass::~AClass(%AClass* %Obj)
  ; Next, continue unwinding:
  unwind
```

Figure 2.4: LLVM code for the C++ example. The handler code specified by `invoke` executes the destructor.

must be executed to throw and recover from exceptions is isolated to the language front-end and language-specific runtime library (so it does not complicate the LLVM representation), however *the exceptional control-flow due to stack unwinding* **is** *encoded within the application code* and therefore exposed in a language-indepenent manner to the optimizer. The C++ exception handling model is very complicated, supporting many related features such as try/catch blocks, checked exception specifications, function try blocks, etc., and reqiring complex semantics for the dynamic lifetime of an exception object. The C++ front-end supports these semantics by generating calls to a simple runtime library.

```
  ; Allocate an exception object
  %t1 = call sbyte* %__llvm_cxxeh_alloc_exc(uint 4)
  %t2 = cast sbyte* %t1 to int*

  ; Construct the thrown value into the memory
  store int 1, int* %t2

  ; ''Throw'' an integer expression, specifying the
  ; exception object, the typeid for the object, and
  ; the destructor for the exception (null for int).
  call void %__llvm_cxxeh_throw(sbyte* %t1,
                                <typeinfo for int>,
                                void (sbyte*)* null)
  unwind              ; Unwind the stack.
```

Figure 2.5: LLVM uses a runtime library for C++ exceptions support but exposes control-flow.

For example, consider the expression '`throw 1`'. This constructs and throws an exception with

integer type. The generated LLVM code is shown in Figure 2.5. The example code illustrates the key feature mentioned above. The runtime handles all of the implementation-specific details, such as allocating memory for exceptions[4]. Second, the runtime functions manipulate the thread-local state of the exception handling runtime, but don't actually unwind the stack. Because the calling code performs the stack unwind, the optimizer has a better view of the control flow of the function without having to perform interprocedural analysis. This allows LLVM to turn stack unwinding operations into direct branches when the unwind target is the same function as the unwinder (this often occurs due to inlining, for example).

Finally, try/catch blocks are implemented in a straight-forward manner, using the same mechanisms and runtime support. Any function call within the try block becomes an `invoke`. Any throw within the try-block becomes a call to the runtime library (as in the example above), followed by an explicit branch to the appropriate catch block. The "catch block" then uses the C++ runtime library to determine if the top-level current exception is of one of the types that is handled in the catch block. If so, it transfers control to the appropriate block, otherwise it calls `unwind` to continue unwinding. The runtime library handles the language-specific semantics of determining whether the current exception is of a caught type.

2.2.5 Plain-text, Binary, and In-memory Representations

The LLVM representation is a *first class language* which defines equivalent textual, binary, and in-memory (i.e., compiler's internal) representations. The instruction set is designed to serve effectively both as a persistent, offline code representation and as a compiler internal representation, with no semantic conversions needed between the two[5]. Being able to convert LLVM code between these representations without information loss makes debugging transformations much simpler, allows test cases to be written easily, and decreases the amount of time required to understand the in-memory representation.

[4]For example, the implementation has to be careful to reserve space for throwing `std::bad_alloc` exceptions.

[5]In contrast, typical JVM implementations convert from the stack-based bytecode language used offline to an appropriate representation for compiler transformations, and some even convert to SSA form for this purpose (e.g., [19]).

2.3 The LLVM Compiler Architecture

The goal of the LLVM compiler framework is to enable sophisticated transformations at link-time, install-time, run-time, and idle-time, by operating on the LLVM representation of a program at all stages. To be practical however, it must be transparent to application developers and end-users, and it must be efficient enough for use with real-world applications. This section describes how the overall system and the individual components are designed to achieve these goals.

2.3.1 High-Level Design of the LLVM Compiler Framework

Figure 2.6 shows the high-level architecture of the LLVM system. Briefly, static compiler front-ends emit code in the LLVM representation, which is combined together by the LLVM linker. The linker performs a variety of link-time optimizations, with a focus on interprocedural techniques. The resulting LLVM code is then translated to native code for a given target at link-time or install-time, and the LLVM code is saved with the native code (alternatively, a JIT compiler can be used). The native code generator inserts light-weight instrumentation to detect frequently executed code regions (currently loop nests and traces, but potentially also functions), and these can be optimized at runtime. The profile data collected at runtime represent the end-user's (not the developer's) runs, and can be used by an offline optimizer to perform aggressive profile-driven optimizations *in the field* during idle-time, tailored to the specific target machine.

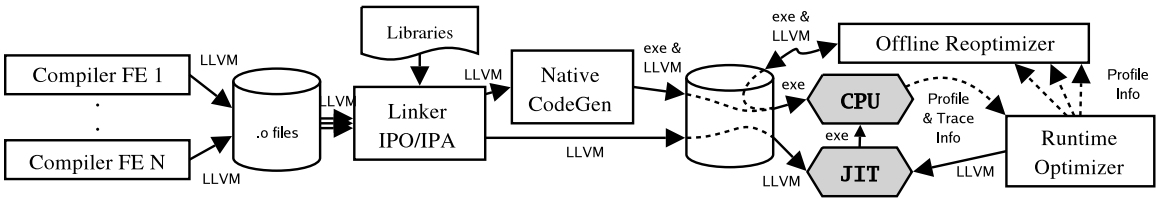


Figure 2.6: LLVM system architecture diagram

This strategy provides five benefits that are not available in the traditional model of static compilation to native machine code. We argued in Section 2.1 that these capabilities are important for lifelong analysis and transformation, and we named them:

1. *persistent program information*,
2. *offline code generation*,
3. *user-based profiling and optimization*,

4. *transparent runtime model*, and
5. *uniform, whole-program compilation.*

These are difficult to obtain simultaneously for at least two reasons. First, offline code generation (#2) normally does not allow optimization at later stages on the higher-level representation instead of native machine code (#1 and #3). Second, lifelong compilation has traditionally been associated only with bytecode-based languages, which do not provide #4 and often not #2 or #5.

In fact, we noted in the Introduction that *no existing compilation approach provides all the capabilities listed above.* Our reasons are as follows:

- Traditional source-level compilers provide #2 and #4, but do not attempt #1, #3 or #5. They do provide interprocedural optimization, but require significant changes to application Makefiles.

- Several commercial compilers provide the additional benefit of #1 and #5 at link-time by exporting their intermediate representation to object files [54, 9, 76] and performing optimizations at link-time. No such system we know of is also capable of preserving its representation for runtime or idle-time use (benefits #1 and #3).

- Higher-level virtual machines like JVM and CLI provide benefit #3 and partially provide #1 (in particular, they focus on runtime optimization, because the need for bytecode verification greatly restricts the optimizations that may be done before runtime [5]). CLI partially provides #5 because it can support code in multiple languages, but any low-level system code and code in non-conforming languages is executed as "unmanaged code". Such code is represented in native form and not in the CLI intermediate representation, so it is not exposed to CLI optimizations. These systems do not provide #2 with #1 or #3 because runtime optimization is generally only possible when using JIT code generation. They do not aim to provide #4, and instead provide a rich runtime framework for languages that match their runtime and object model, e.g., Java and C#. Omniware [2] provides #5 and most of the benefits of #2 (because, like LLVM, it uses a low-level represention that permits extensive static optimization), but at the cost of not providing information for high-level analysis and optimization (i.e., #1). It does not aim to provide #3 or #4.

- Transparent binary runtime optimization systems like Dynamo and the runtime optimizers in Transmeta processors provide benefits #2, #4 and #5, but they do not provide #1. They provide benefit #3 only at runtime, and only to a limited extent because they work only on native binary code, limiting the optimizations they can perform.

- Profile Guided Optimization for static languages provide benefit #3 at the cost of not being transparent (they require a multi-phase compilation process). Additionally, PGO suffers from three problems: (1) Empirically, developers are unlikely to use PGO, except when compiling benchmarks. (2) When PGO *is* used, the application is tuned to the behavior of the training run. If the training run is not representative of the end-user's usage patterns, performance may not improve and may even be hurt by the profile-driven optimization. (3) The profiling information is completely static, meaning that the compiler cannot make use of phase behavior in the program or adapt to changing usage patterns.

There are also significant limitations of the LLVM strategy. First, language-specific optimizations must be performed in the front-end before generating LLVM code. LLVM is *not* designed to represent source languages types or features directly. Second, it is an open question whether languages requiring sophisticated runtime systems such as Java can benefit directly from LLVM. We are currently exploring the potential benefits of implementing higher-level virtual machines such as JVM or CLI on top of LLVM.

The subsections below describe the key components of the LLVM compiler architecture, emphasizing design and implementation features that make the capabilities above practical and efficient.

2.3.2 Compile-Time: External Front-end and Static Optimizer

External static LLVM compilers (referred to as front-ends) translate source-language programs into the LLVM virtual instruction set. Each static compiler can perform three key tasks, of which the first and third are optional: (1) Perform language-specific optimizations, e.g., optimizing closures in languages with higher-order functions. (2) Translate source programs to LLVM code, synthesizing as much useful LLVM type information as possible, especially to expose pointers, structures, and arrays. (3) Invoke LLVM passes for global or interprocedural optimizations at the module level. The LLVM optimizations are built into libraries, making it easy for front-ends to use them.

The front-end does not have to perform SSA construction. Instead, variables can be allocated on the stack (which is not in SSA form), and the LLVM stack promotion and scalar expansion passes can be used to build SSA form effectively. Stack promotion converts stack-allocated scalar values to SSA registers if their address does not escape the current function, inserting ϕ functions as necessary to preserve SSA form. Scalar expansion precedes this and expands local structures to scalars wherever possible, so that their fields can be mapped to SSA registers as well.

Note that many "high-level" optimizations are not really language-dependent, and are often special cases of more general optimizations that may be performed on LLVM code. For example, both virtual function resolution for object-oriented languages (described in Section 7) and tail-recursion elimination which is crucial for functional languages can be done in LLVM. In such cases, it is better to extend the LLVM optimizer to perform the transformation, rather than investing effort in code which only benefits a particular front-end. This also allows the optimizations to be performed throughout the lifetime of the program.

2.3.3 Linker & Interprocedural Optimizer

Link time is the first phase of the compilation process where most[6] of the program is available for analysis and transformation. As such, link-time is a natural place to perform aggressive interprocedural optimizations across the entire program. The link-time optimizations in LLVM operate on the LLVM representation directly, taking advantage of the semantic information it contains. LLVM currently includes a number of interprocedural analyses, such as Data Structure Analysis (Chapter 3), call graph construction, Mod/Ref analysis, and interprocedural transformations like inlining, dead global elimination, dead argument elimination, dead type elimination, constant propagation, array bounds check elimination [82], simple structure field reordering, and Automatic Pool Allocation (Chapter 5).

The design of the compile- and link-time optimizers in LLVM permit the use of a well-known technique for speeding up interprocedural analysis. At compile-time, interprocedural summaries can be computed for each function in the program and attached to the LLVM bytecode. The link-time interprocedural optimizer can then process these interprocedural summaries as input instead

[6]Note that shared libraries and system libraries may not be available for analysis at link time, or may be compiled directly to native code.

of having to compute results from scratch. This technique can dramatically speed up incremental compilation when a small number of translation units are modified [18]. Note that this is achieved without building a program database or deferring the compilation of the input source code until link-time.

2.3.4 Offline or JIT Native Code Generation

Before execution, a code generator is used to translate from LLVM to native code for the target platform (we currently support the X86, PowerPC, Sparc V8/V9, and Alpha architectures), in one of two ways. In the first option, the code generator is run statically at link time or install time, to generate high performance native code for the application, using possibly expensive code generation techniques. If the user decides to use the post-link (runtime and offline) optimizers, a copy of the LLVM bytecode for the program is included into the executable itself. In addition, the code generator can insert light-weight instrumentation into the program to identify frequently executed regions of code.

Alternatively, a Just-In-Time Execution Engine can be used which invokes the appropriate code generator at runtime, translating one function at a time for execution (or uses the portable (but slow) LLVM interpreter if no native code generator is available). The JIT translator can also insert the same instrumentation as the offline code generator.

2.3.5 Runtime Path Profiling & Reoptimization

One of the goals of the LLVM project is to develop a new strategy for runtime optimization of ordinary applications. Although that work is outside the scope of this thesis, we briefly describe the strategy and its key benefits.

As a program executes, the most frequently executed execution paths are identified through a combination of offline and online instrumentation [124]. The offline instrumentation (inserted by the native code generator) identifies frequently executed loop regions in the code. When a hot loop region is detected at runtime, a runtime instrumentation library instruments the executing native code to identify frequently-executed paths within that region. Once hot paths are identified, we duplicate the original LLVM code into a trace, perform LLVM optimizations on it, and then

regenerate native code into a software-managed trace cache. We then insert branches between the original code and the new native code.

The strategy described here is powerful because it combines the following three characteristics: (a) Native code generation can be performed ahead-of-time using sophisticated algorithms to generate high-performance code. (b) The native code generator and the runtime optimizer can work together since they are both part of the LLVM framework, allowing the runtime optimizer to exploit support from the code generator (e.g., for instrumentation and simplifying transformations). (c) The runtime optimizer can use high-level information from the LLVM representation to perform sophisticated runtime optimizations.

We believe these three characteristics together represent one "optimal" design point for a runtime optimizer because they allow the best choice in three key aspects: high-quality initial code generation (offline rather than online), cooperative support from the code-generator, and the ability to perform sophisticated analyses and optimizations (using LLVM rather than native code as the input).

2.3.6 Offline Reoptimization with End-user Profile Information

Because the LLVM representation is preserved permanently, it enables transparent offline optimization of applications during idle-time on an end-user's system. Such an optimizer is simply a modified version of the link-time interprocedural optimizer, but with a greater emphasis on profile-driven and target-specific optimizations.

An offline, idle-time reoptimizer has several key benefits. First, as noted earlier, unlike traditional profile-guided optimizers (i.e., compile-time or link-time ones), it can use profile information gathered from end-user runs of the application. It can even reoptimize an application multiple times in response to changing usage patterns over time (or optimize differently for users with differing patterns). Second, it can tailor the code to detailed features of a single target machine, whereas traditional binary distributions of code must often be run on many different machine configurations with compatible architectures and operating systems. Third, unlike the runtime optimizer (which has both the previous benefits), it can perform much more aggressive optimizations because it is run offline.

Nevertheless, runtime optimization can further improve performance because of the ability to perform optimizations based on runtime values as well as path-sensitive optimizations (which can cause significant code growth if done aggressively offline), and to adaptively optimize code for changing execution behavior within a run. For dynamic, long-running applications, therefore, the runtime and offline reoptimizers could coordinate to ensure the highest achievable performance.

2.4 Applications and Experiences

Sections 2.2 and 2.3 describe the design of the LLVM code representation and compiler architecture. In this section, we evaluate this design in terms of three categories of issues: (a) the characteristics of the representation; (b) the speed of performing whole-program analyses and transformations in the compiler; and (c) illustrative uses of the LLVM system for challenging compiler problems (such as macroscopic analyses and transformations), focusing on how the novel capabilities in LLVM benefit these uses.

2.4.1 Representation Issues

We evaluate three important characteristics of the LLVM representation. First, a key aspect of the representation is the language-independent type system. Does this type system provide any useful information when it can be violated with casts? Second, how do high-level language features map onto the LLVM type system and code representation? Third, how large is the LLVM representation when written to disk?

What value does type information provide?

Reliable type information for programs can enable the optimizer to perform aggressive transformations that would be difficult otherwise, such as reordering two fields of a structure or optimizing memory management (as described throughout this thesis). As noted in Section 2.2.2, however, type information in LLVM is not reliable and some analysis (typically including a pointer analysis) must check the declared type information before it can be used. A key question is how much *reliable* type information is available in programs compiled to LLVM?

Data Structure Analysis (DSA, described in Chapter 3 is a flow-insensitive, field-sensitive and context-sensitive points-to analysis included with LLVM. DSA serves as an important host for all of the macroscopic techniques described in this thesis. As part of its analysis, DSA extracts LLVM types for a subset of memory objects in the program. It does this by using declared types in the LLVM code as speculative type information, and checks conservatively whether memory accesses to an object are consistent with those declared types[7], without having to perform type-inference.

Section 3.4.3 describes the use of DSA to verify the type information provided by LLVM for a suite of C, C++, and FORTRAN programs, counting the fraction of load/store operations and store operations for which reliable type information about the accessed objects is available using DSA. It shows that a vast amount of type information is available for C programs, despite the fact that the language permits all sorts of non-type-safe behavior.

It is important to note that similar results would be very difficult to obtain if LLVM had been an untyped representation. Intuitively, checking that declared types are respected is much easier (and requires less analysis time) than inferring those types for structure and array types in a low-level code representation. As an example, an earlier version of the LLVM C front-end was based on GCC's RTL internal representation, which provided little useful type information, and both DSA and pool allocation were much less effective. Our new C/C++ front-end is based on the GCC Abstract Syntax Tree representation, which makes much more type information available.

How do high-level features map onto LLVM?

LLVM is a much lower level representation than source code for standard source languages. Even C, which itself is quite low-level, has many features which must be lowered by a compiler targeting LLVM. For example, complex numbers, structure copies, unions, bit-fields, variable sized arrays, and `setjmp/longjmp` all must be lowered by an LLVM C compiler (and all are supported by llvm-gcc). In order for the representation to support effective analyses and transformations, the mapping from source-language features to LLVM should capture the high-level operational behavior as cleanly as possible.

We discuss this issue by using C++ as an example, since it is the richest language for which we

[7]DSA is quite aggressive: it can often extract type information for objects stored into and loaded out of "generic" `void*` data structure, despite the casts to and from `void*`.

have a fully-functional front-end. We believe that all the complex, high-level features of C++ are expressed clearly in LLVM, allowing their behavior to be effectively analyzed and optimized:

- String constants (e.g. "hello world") are lowered to constant, preinitialized, LLVM global variables.

- Implicit calls (e.g. copy constructors) and parameters (e.g. 'this' pointers) are made explicit.

- Templates are fully instantiated by the C++ front end before LLVM code is generated. Languages with true polymorphic types would be expanded into equivalent code using non-polymorphic types in LLVM.

- Base classes are expanded into nested structure types. For this C++ fragment:

```
class base1 { int Y; };
class base2 { float X; };
class derived : base1, base2 { short Z; };
```

  the LLVM type for class `derived` is '`{ {int}, {float}, short }`'. If the classes have virtual functions, a v-table pointer would also be included and initialized at object allocation time to point to the virtual function table, described below.

- A virtual function table is represented as a global, *constant* array of typed function pointers, plus the type-id object for the class. With this representation, virtual method call resolution can be performed by the LLVM optimizer as effectively as by a typical source compiler (more effectively if the source compiler uses only per-module instead of cross-module pointer analysis).

- C++ exceptions are lowered to the '`invoke`' and '`unwind`' instructions as described in Section 2.2.4, exposing exceptional control flow in the CFG. In fact, having this information available at link time enables LLVM to use an interprocedural analysis to eliminate unused exception handlers. This optimization is much less effective if done on a per-module basis in a source-level compiler.

We believe that similarly clean LLVM implementations exist for most constructs in other language families like Scheme, the ML family, SmallTalk, Java and Microsoft CLI. We have implementation experience with prototype MS CLI and Java virtual machines that indicate likely success.

How compact is the LLVM representation?

Since code for the compiled program is stored in the LLVM representation throughout its lifetime, it is important that it not be too large. The flat, three-address form of LLVM is well suited for a simple linear layout, with most instructions requiring only a single 32-bit word each in the file. Figure 2.7 shows the size of LLVM files for SPEC CPU2000 executables after linking, compared to native X86 and 32-bit Sparc executables compiled by GCC 3.3 at optimization level -O3[8].

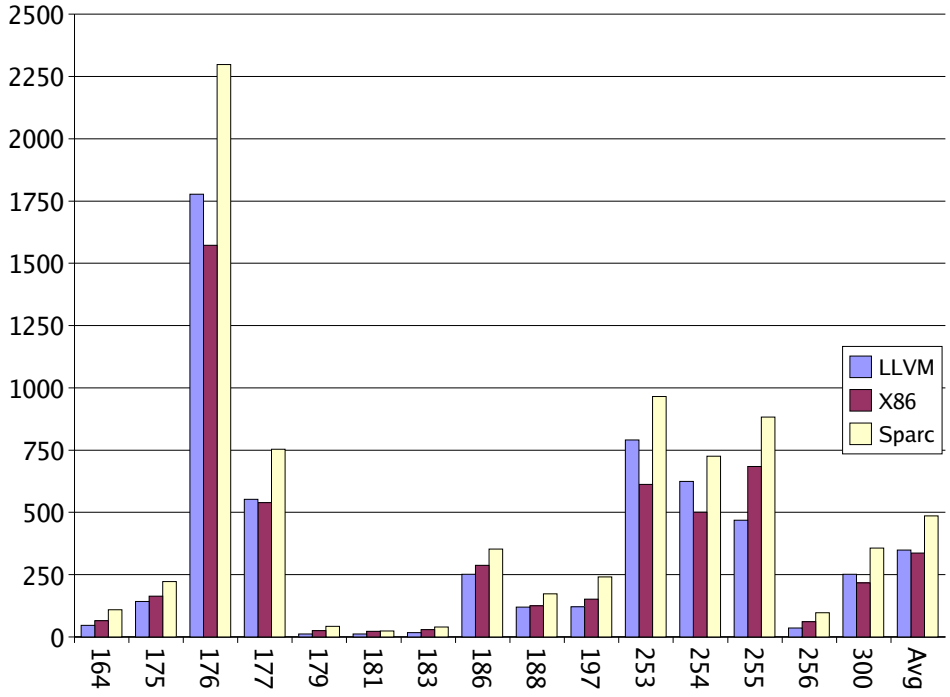


Figure 2.7: Executable sizes for LLVM, X86, Sparc (in KB)

The figure shows that LLVM code is about the same size as native X86 executables (a dense, variable-size instruction set), and significantly smaller than SPARC (a traditional 32-bit instruction RISC machine). We believe this is a very good result given that LLVM encodes an infinite register

[8]Note that LLVM compresses bytecode files with bzip2 by default. These numbers are collected with this compression feature turned off. LLVM can read both compressed and non-compressed bytecode files natively.

set, rich type information, control flow information, and data-flow (SSA) information that native executables do not.

Currently, large programs are encoded less efficiently than smaller ones because they have a larger set of register values available at any point, making it harder to fit instructions into a 32-bit encoding. When an instruction does not fit into a 32-bit encoding, LLVM falls back on a 64-bit or larger encoding, as needed. Though it would be possible to make the fall back case more efficient, we have not attempted to do so.

How fast is LLVM?

An important aspect of LLVM is that the low-level representation enables *efficient* analysis and transformation, because of the small, uniform instruction set, the explicit CFG and SSA representations, and careful implementation of data structures. This speed is important for uses "late" in the compilation process (i.e., at link-time or run-time). In order to provide a sense for the speed of LLVM, Figure 2.8 shows the table of runtimes for several interprocedural optimizations. All timings were collected on a 3.06GHz Intel Xeon processor. The LLVM compiler system was compiled using the GCC 3.3 compiler at optimization level -O3.

Benchmark	DGE	DAE	inline	GCC
164.gzip	0.0018	0.0063	0.0127	1.937
175.vpr	0.0096	0.0082	0.0564	5.804
176.gcc	0.0496	0.1058	0.6455	55.436
177.mesa	0.0051	0.0312	0.0788	20.844
179.art	0.0002	0.0007	0.0085	0.591
181.mcf	0.0010	0.0007	0.0174	1.193
183.equake	0.0000	0.0009	0.0100	0.632
186.crafty	0.0016	0.0162	0.0531	9.444
188.ammp	0.0200	0.0072	0.1085	5.663
197.parser	0.0021	0.0096	0.0516	5.593
253.perlbmk	0.0137	0.0439	0.8861	25.644
254.gap	0.0065	0.0384	0.1317	18.250
255.vortex	0.1081	0.0539	0.2462	20.621
256.bzip2	0.0015	0.0028	0.0122	1.520
300.twolf	0.0712	0.0152	0.1742	11.986

Figure 2.8: Interprocedural optimization timings (in seconds)

The table includes numbers for several transformations: **DGE** (aggressive[9] Dead Global variable and function Elimination), **DAE** (aggressive Dead Argument (and return value) Elimination), and

[9] "Aggressive" DCEs assume objects are dead until proven otherwise, allowing dead objects with cycles to be deleted.

inline (a function integration pass). All these interprocedural optimizations work on the whole program at link-time. In addition, they spend most of their time traversing and modifying the code representation directly, so they reflect the costs of processing the representation. As a reference for comparison, the **GCC** column indicates the total time the GCC 3.3 compiler takes to compile the program at -O3.

We find that in all cases, the optimization time is substantially less than that to compile the program with GCC, despite the fact that GCC does *no* cross module optimization, and very little interprocedural optimization within a translation unit. In addition, the interprocedural optimizations scale mostly linear with the number of transformations they perform. For example, DGE eliminates 331 functions and 557 global variables (which include string constants) from 255.vortex, DAE eliminates 103 arguments and 96 return values from 176.gcc, and 'inline' inlines 1368 functions (deleting 438 which are no longer referenced) in 176.gcc.

2.4.2 Example Applications of LLVM

Finally, to illustrate the capabilities provided by the compiler framework, we briefly describe four examples of how LLVM has been used for widely varying compiler problems, emphasizing some of the novel capabilities described in the introduction.

Projects using LLVM as a general compiler infrastructure

As noted earlier, LLVM has served as the host for many varied compiler techniques. The most aggressive of these are Data Structure Analysis (DSA) and Automatic Pool Allocation, which analyze and transform programs in terms of their logical data structures (as described in the rest of this thesis). These techniques inherit a few significant benefits from LLVM, especially, (a) these techniques are only effective if most of the program is available, i.e., at link-time; (b) type information is crucial for their effectiveness, especially pointers and structures; (c) the techniques are source-language independent; and (d) SSA significantly improves the precision of DSA, which is flow-insensitive.

Other researchers not affiliated with our group have been actively using or exploring the use of the LLVM compiler framework, in a number of different ways. These include using LLVM as

an intermediate representation for binary-to-binary transformations, as a compiler back-end to support a hardware-based trace cache and optimization system, as a basis for runtime optimization and adaptation of Grid programs, as an implementation platform for several novel programming languages, to JIT compile shaders in a photo-realistic renderer, etc.

As of this writing, LLVM has shipped 5 major releases roughly once every three months. The rapidly growing user base building around the compiler is a testament to its flexibility and adaptability.

SAFECode: A safe low-level representation and execution environment

SAFECode is a "safe" code representation and execution environment, based on a type-safe subset of LLVM. The goal of the work is to enforce memory safety of programs in the SAFECode representation through static analysis, by using a variant of automatic pool allocation instead of garbage collection [48], and using extensive interprocedural static analysis to minimize runtime checks [82, 48].

The SAFECode system exploits nearly all capabilities of the LLVM framework, except runtime optimization. It directly uses the LLVM code representation, which provides the ability to analyze C and C++ programs, which is crucial for supporting embedded software, middle-ware, and system libraries. SAFECode relies on the type information in LLVM (with no syntactic changes) to check and enforce type safety. It relies on the array type information in LLVM to enforce array bounds safety, and uses interprocedural analysis to eliminate runtime bounds checks in many cases [82]. It uses interprocedural safety checking techniques, exploiting the link-time framework to retain the benefits of separate compilation (a key difficulty that led previous such systems to avoid using interprocedural techniques [44, 63]).

External ISA design for Virtual Instruction Set Computers

Virtual Instruction Set Computers [125, 43, 3] are processor designs that use two distinct instruction sets: an externally visible, virtual instruction set (V-ISA) which serves as the program representation for all software, and a hidden implementation-specific instruction set (I-ISA) that is the actual hardware ISA. A software translator co-designed with the hardware translates V-ISA code

to the I-ISA transparently for execution, and is the only software that is aware of the I-ISA. This translator is essentially a sophisticated, implementation-specific back-end compiler.

In recent work, we argued that an extended version of the LLVM instruction set could be a good choice for the external V-ISA for such processor designs [3]. We proposed a novel implementation strategy for the virtual-to-native translator that enables offline code translation and caching of translated code in a completely OS-independent manner.

That work *exploits* the important features of the instruction set representation, and extends it to be suitable as a V-ISA for hardware. The fundamental benefit of LLVM for this work is that the LLVM code representation is low-level enough to represent arbitrary external software (including operating system code), yet provides rich enough information to support sophisticated compiler techniques in the translator. A second key benefit is the ability to do both offline and online translation, which is exploited by the OS-independent translation strategy.

2.5 Related Work

We focus on comparing LLVM with three classes of previous work: other virtual-machine-based compiler systems, research on typed assembly languages, and link-time or dynamic optimization systems.

As noted in the introduction, the goals of LLVM are complementary to those of higher-level language virtual machines such as SmallTalk, Self, JVM, and the managed mode of Microsoft CLI. High-level virtual machines such as these require a particular object model and runtime system for use. This implies that they can provide higher-level type information about the program, but are not able to support languages that do not match their design (even object-oriented languages such as C++). Additionally, programs in these representations (except CLI) are required to be type-safe. This is important for supporting mobile code, but makes these virtual machines insufficient for non-type-safe languages and for low-level system code. It also significantly limits the amount of optimization that can be done before runtime because of the need for bytecode verification.

The Microsoft CLI virtual machine has a number of features that distinguish it from other high-level virtual machines, including explicit support for a wide range of features from multiple languages, language interoperability support, non-type-safe code, and "unmanaged" execution

mode. Unmanaged mode allows CLI to represent code in arbitrary languages, including those that do not conform to its type system or runtime framework, e.g., ANSI-standard C++ [96]. However, code in unmanaged mode is not represented in the CLI intermediate representation (MSIL), and therefore is not subject to dynamic optimization in CLI. In contrast, LLVM allows code from arbitrary languages to be represented in a uniform, rich representation and optimized throughout the lifetime of the code. A second key difference is that LLVM lacks the interoperability features of CLI but also does not require source-languages to match the runtime and object model for interoperability. Instead, it requires source-language compilers to manage interoperability, but then allows all such code to be exposed to LLVM optimizers at all stages.

The Omniware virtual machine [2] is closer to LLVM, because they use an abstract low-level RISC architecture and can support arbitrary code (including non-type-safe code) from any source language. However, the Omniware instruction set lacks the higher-level type information of LLVM. In fact, it allows (and requires) source compilers to choose data layouts, perform address arithmetic, and perform register allocation (to a small set of virtual registers). All these features make it difficult to perform any sophisticated analysis on the resulting Omniware code. These differences from LLVM arise because the goals of their work are primarily to provide code mobility and safety, not a basis for lifelong code optimization. Their virtual machine compiles Omniware code to native code at runtime, and performs only relatively simple optimizations plus some stronger machine-dependent optimizations.

Kistler and Franz describe a compilation architecture for performing optimization in the field, using simple initial load-time code generation, followed by profile-guided runtime optimization [81]. Their system targets the Oberon language, uses Slim Binaries [56] as its code representation, and provides type safety and memory management similar to other high-level virtual machines. They do not attempt to support arbitrary languages or to use a transparent runtime system, as LLVM does. They also do not propose doing static or link-time optimization.

There has been a wide range of work on typed intermediate representations. Functional languages often use strongly typed intermediate languages (e.g. [123]) as a natural extension of the source language. Projects on typed assembly languages (e.g., TAL [99] and LTAL [24]) focus on preserving high-level type information and type safety during compilation and optimizations. The

SafeTSA [5] representation is a combination of type information with SSA form, which aims to provide a safe but more efficient representation than JVM bytecode for Java programs. In contrast, the LLVM virtual instruction set does not attempt to preserve type safety of high-level languages, to capture high-level type information from such languages, or to enforce code safety directly (though it can be used to do so [48]). Instead, the goal of LLVM is to enable sophisticated analyses and transformations beyond static compile time.

There have been attempts to define a unified, generic, intermediate representation. These have largely failed, ranging from the original UNiversal Computer Oriented Language [127] (UNCOL), which was discussed but never implemented, to the more recent Architecture and language Neutral Distribution Format [7] (ANDF), which was implemented but has seen limited use. These unified representations attempt to describe programs at the AST level, by including features from all supported source languages. LLVM is much less ambitious and is more like an assembly language: it uses a small set of types and low-level operations, and the "implementation" of high-level language features is described in terms of these types. In some ways, LLVM simply appears as a strict RISC architecture.

Several systems perform interprocedural optimization at link-time. Some operate on assembly code for a given processor [101, 126, 34, 110] (focusing primarily on machine-dependent optimizations), while others export additional information from the static compiler, either in the form of an IR or annotations [139, 54, 9, 76]. None of these approaches attempt to support optimization at runtime or offline after software is installed in the field, and it would be difficult to directly extend them to do so.

There have also been several systems that perform transparent runtime optimization of native code [10, 50, 43]. These systems inherit all the challenges of optimizing machine-level code [101] in addition to the constraint of operating under the tight time constraints of runtime optimization. In contrast, LLVM aims to provide type, dataflow (SSA) information, and an explicit CFG for use by runtime optimizations. For example, our online tracing framework (Section 2.3.5) directly exploits the CFG at runtime to perform limited instrumentation of hot loop regions. Finally, none of these systems supports link-time, install-time, or offline optimizations, with or without profile information.

2.6 Conclusion

This chapter described LLVM, a system for performing lifelong code analysis and transformation, while remaining transparent to programmers. The system uses a low-level, typed, SSA-based instruction set as the persistent representation of a program, but without imposing a specific runtime environment. The LLVM representation is language independent, allowing all the code for a program, including system libraries and portions written in different languages, to be compiled and optimized together. The LLVM compiler framework is designed to permit optimization at all stages of a software lifetime, including extensive static optimization, online optimization using information from the LLVM code, and idle-time optimization using profile information gathered from programmers in the field. The current implementation includes a powerful link-time global and interprocedural optimizer, a low-overhead tracing technique for runtime optimization, and Just-In-Time and static code generators.

We showed experimentally and based on experience that LLVM (with Data Structure Analysis from Chapter 3) makes available extensive type information even for C programs, which can be used to safely perform a number of aggressive transformations that would normally be attempted only on type-safe languages in source-level compilers. We also showed that the LLVM representation is comparable in size to X86 machine code and about 25% smaller than SPARC code on average, despite capturing much richer type information as well as an infinite register set in SSA form. Finally, we gave several examples of whole-program optimizations that are very efficient to perform on the LLVM representation. A key question we are exploring currently is whether high-level language virtual machines can be implemented effectively on top of the LLVM runtime optimization and code generation framework.

Chapter 3

Data Structure Analysis

Alias analysis for programs with complex pointer-based data structures has been most successful at guiding traditional low-level memory optimizations. These transformations rely on disambiguating pairs of memory references and on identifying local and interprocedural side-effects of statements. In contrast, there has been much less success with transformations that apply to entire *instances* of *data structures* such as a lists, heaps, or graphs. Many reasons exist for this disparity, including the possibility of non-type-safe memory accesses in common programming languages (e.g., C and C++), and the potentially high cost of an analysis that can distinguish different instances of a logical data structure.

Enabling such analyses and transformations requires some powerful analysis capabilities:

1. **Full Context-Sensitivity**: Identifying distinct *instances* of data structures requires the analysis algorithm to distinguish between heap objects created via different call paths in a program (i.e., naming objects by their call paths), because data structures are often created with common library functions. Even many partially context-sensitive algorithms do not attempt to distinguish heap objects by call paths [51, 143, 53, 138, 42], which makes them unable to detect this key property. On the other hand, naïve approaches to full context sensitivity can easily lead to an explosion in the size of the heap representation (because the number of call paths is often exponential in the size of the program), and can make recursion difficult to handle.

2. **Field-Sensitivity**: Identifying the internal connectivity pattern of a data structure requires distinguishing the points-to properties of different structure fields. Such "field-sensitivity" is

often supported by analyses targeting languages that are type-safe but is difficult to support efficiently (if at all) in non-type-safe languages (e.g., see [128, 92]).

3. **Explicit Heap Model:** Analyzing memory objects requires constructing an explicit model of the memory in the program, including objects not directly necessary for identifying aliases. Some common alias analysis algorithms (e.g., Steensgaard's [129] and Andersen's [6] algorithms) build an explicit heap representation, but do not provide any context-sensitivity. Other, more powerful analyses only record alias pairs to determine pointer aliasing properties [51, 25, 73, 140]. Retaining both capabilities is challenging due to the potential for the heap model to grow to be very large.

Practical alias and pointer analysis algorithms have not attempted to provide the combination of properties described above, or are not fast enough for realistic use in a commercial compiler (requiring minutes to hours of analysis time for medium size programs). If compile time is no issue, "shape analysis" algorithms are powerful enough to provide this information and more (e.g., enough to identify a particular structure as a "linked-list" or "binary tree" [60, 117]). Shape analysis, however, has so far not proven practical for use in commercial optimizing compilers due to its intractable scalability with current algorithms.

In this work, we present an analysis algorithm called **Data Structure Analysis**, which is the key foundation for all of the macroscopic techniques described in this thesis. The algorithm aims to lie somewhere between traditional pointer analyses and more powerful shape analysis algorithms in power, while being as fast as traditional aggressive alias analyses. It provides the three required capabilities listed above and it supports the full generality of C programs, including type-unsafe code, incomplete programs, function pointers, recursion, and `setjmp/longjmp`. We believe it is efficient and scalable enough for use in commercial compilers.

We show that the theoretical worst case time and memory complexity are $\Theta(n\alpha(n) + k\alpha(k)e)$, and $\Theta(fk)$, where n, k, e, and f denote the number of instructions, the maximum number of nodes in a data structure graph for a single procedure, the number of edges in the call graph, and the total number of functions. In practice, k is small, typically on the order of a hundred nodes or less, even in large programs.

We evaluate the algorithm on 35 C programs, showing that the algorithm is extremely efficient

in practice (in both performance and memory consumption). This includes programs that contain complex heap structures, recursion, and function pointers. For example, it requires less than 3.5 seconds of analysis time and about 19MB of memory to analyze `176.gcc`, a program consisting of over 222,000 lines of code. Overall, we believe the broader implication of our work is to show that a fully context sensitive analysis as described here can be practical for significant, large, real-world programs.

The remainder of this chapter is organized as follows: Section 3.1 introduces the data structure graph representation and its semantics. Section 3.2 describes the algorithms used to construct the graphs used by the analysis. Section 3.3 describes important engineering and implementation issues that are critical for making the analysis efficient in practice. Section 3.4 evaluates the analysis time, memory usage, and type information provided by the compiler (a study of DSA precision is included in Chapter 4). Section 3.5 contrasts our work with prior art in the field. Finally, Section 3.6 summarizes the key contributions and results of Data Structure Analysis.

3.1 The Data Structure Graph

Data Structure Analysis computes a graph we call the Data Structure Graph (DS graph) for each function in a program, summarizing the memory objects accessible within the function along with their connectivity patterns. Each DS graph node represents a (potentially unbounded) set of dynamic memory objects and distinct nodes represent disjoint sets of objects, i.e., the graph is a finite, static partitioning of the memory objects. Because we use a unification-based approach, all dynamic objects which may be pointed to by a single static pointer variable or field (in some context) are represented as a single node in the graph.

Some assumptions about the input program representation are necessary for describing our graph representation; other details are described in Section 3.2.2. In practice we perform all analysis on the LLVM representation described in Section 2.2. However, the requirements we assume are provided by many systems, so we describe the relevant aspects that DSA depends on below.

We assume that input programs have a simple type system with structural type equivalence, having primitive integer and floating point types of predefined sizes, plus four derived types: pointers, structures (i.e., record types), arrays, and functions. We assume (as in the C language) that

only explicit pointer types and integer types of the same size or larger can directly encode a pointer value, and call these *pointer-compatible* types (other values are handled very conservatively in the analysis). For any type τ, $fields(\tau)$ returns a set of field names for the fields of τ, which is a single degenerate field name if τ is a scalar type (field names are assumed to be unique to a type). An array type of known size k may be represented either as a structure with k fields or by a single field; an unknown-size array is always represented as the latter. Other assumptions about the input program representation are described in Section 3.2.2.

We also assume a load/store program representation in which virtual registers and memory locations are distinct. In our representation it is not possible to take the address of a virtual register, so address taken variables must live in memory. Additionally, virtual registers can only represent scalar variables (i.e., integer, floating point, or pointer). Structures, arrays, and functions are strictly memory objects and are accessed only through load, store, and call instructions. All arithmetic operations operate on virtual registers. Memory is partitioned into heap objects (allocated via a `malloc` instruction), stack objects (allocated via an explicit stack allocation instruction named `alloca`, similar to malloc), global objects (global variables and functions), and unknown objects.

The DS graph for a function is a finite directed multigraph represented as a tuple $DSG(F) = \langle N, E, E_V, C \rangle$, where:

- N is a set of nodes, called "DS nodes". DS nodes have several attributes described in Section 3.1.1 below.

- E is a set of edges in the graph. Formally, E is a function of type $\langle n_s, f_s \rangle \rightarrow \langle n_d, f_d \rangle$, where $n_s, n_d \in N$, $f_s \in fields(T(n_s))$ and $f_d \in fields(T(n_d))$, and $T(n)$ denotes type information computed for the objects of n as explained below. E is a function because a source field can have only a single outgoing edge. Note that the source and target of an edge are both *fields* of a DS node.

- E_V is a function of type $vars(f) \rightarrow \langle n, f \rangle$, where $vars(f)$ is the set of virtual registers in function f. Conceptually, $E_V(v)$ is an edge from register v to the target field $\langle n, f \rangle$ pointed to by v, if v is of pointer-compatible type.

- C is a set of "call nodes" in the graph (seperate from N), which represent unresolved call sites in the context of the current function. Each call node $c \in C$ is a $k+2$ tuple: $(r, f, a_1, \ldots, a_k)$, where every element of the tuple is a node-field pair $\langle n, f \rangle$. r and f respectively denote the value returned by the call (if it is pointer-compatible) and the function(s) being called. $a_1 \ldots a_k$ denote the pointer-compatible values passed as arguments to the call (other arguments are not represented). Conceptually, each tuple element can also be regarded as a points-to edge in the graph.

```
typedef struct list {
  struct list *Next;
  int Data;
} list;

int Global = 10;

void do_all(list *L, void (*FP)(int*)) {
  do { FP(&L->Data);
       L = L->Next;
  } while(L);
}

void addG(int *X) {
  (*X) += Global;
}

void addGToList(list *L) {
  do_all(L, addG);
}

list *makeList(int Num) {
  list *New = malloc(sizeof(list));
  New->Next = Num ? makeList(Num-1) : 0;
  New->Data = Num;
  return New;
}

int main() {    /* X & Y lists are disjoint */
  list *X = makeList(10);
  list *Y = makeList(100);
  addGToList(X);
  Global = 20;
  addGToList(Y);
}
```

Figure 3.1: C code for running example

To illustrate the DS graphs and the analysis algorithm, we use the code in Figure 3.1 as a running example. This example creates and traverses two disjoint linked lists, using iteration,

recursion, function pointers, a pointer to a subobject, and a global variable reference. Despite the complexity of the example, Data Structure Analysis is able to prove that the two lists `X` and `Y` are disjoint (the final DS graph computed for `main` is shown in Figure 3.10).

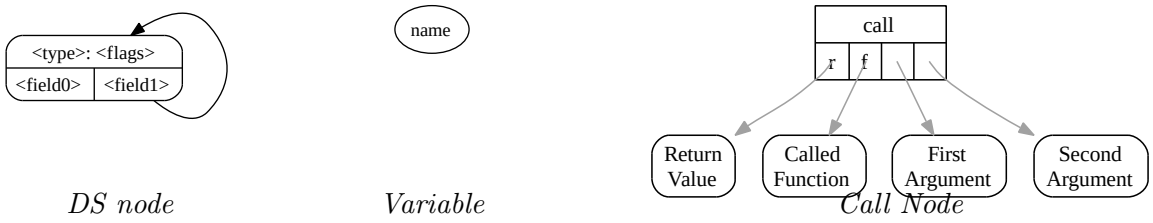


Figure 3.2: Graph Notation

To illustrate the DS graphs computed by various stages of our algorithm, we render DS graphs using the graphical notation shown in Figure 3.2. Figure 3.3 shows an example graph computed for the `do_all` and `addG` functions, before any interprocedural information is applied. The figure includes an example of a call node, which (in this case) calls the function pointed to by `FP`, passing the memory object pointed to by `L` as an argument, and ignores the return value of the call.

3.1.1 Graph Nodes and Fields

The DS nodes in a DS graph are responsible for representing information about a set of memory objects corresponding to that node. Each node n has three pieces of information associated with it:

- $T(n)$ identifies a type for the memory objects represented by n. Section 10 describes how this is computed for nodes representing multiple incompatible memory objects.
- $G(n)$ represents a (possibly empty) set of global objects, namely, all those represented by node n. Note that functions are treated as global objects.
- $flags(n)$ is a set of flags associated with node n. There are eight possible flags (H,S,G,U, M,R, C and O), defined below.

The type information $T(n)$ determines the number of fields and outgoing edges in a node. A node can have one outgoing edge for each pointer-compatible field in $T(n)$. An incoming edge can point to an arbitrary field of the node (e.g., the "`&L->Data`" temporary in Figure 3.3 points to the

integer field), but not to any other byte offset. We describe how type-unsafe code using pointers to arbitrary byte offests are handled below.

The globals $G(n)$ represented by each node can be used to find the targets of function pointers, both by clients of Data Structure Analysis and to incrementally construct the call-graph during the analysis.

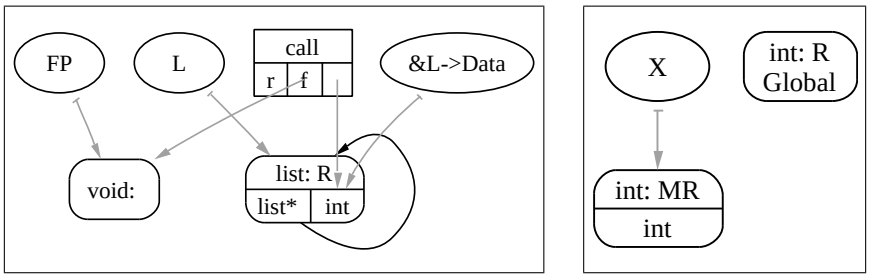


Figure 3.3: Local DSGraphs for `do_all` and `addG`

Memory Allocation Class Flags: H, S, G, U

The **'H'**, **'S'**, **'G'** and **'U'** flags in $flags(n)$ are used to distinguish four classes of memory objects: **H**eap-allocated, **S**tack-allocated, **G**lobals (which include functions), and **U**nknown objects. Multiple flags may be present in a single DS node, if, for example, analysis finds a pointer which may point to either a heap object or a stack object. Memory objects are marked as Unknown when the instruction creating it is not identifiable, e.g., when a constant value is cast to a pointer value (for example, to access a memory-mapped hardware device), or when unanalyzable address arithmetic is found (these cases occur infrequently in portable programs). Nodes representing objects created in an external, unanalyzed function are *not* marked **'U'**, but are treated as "missing information" as described below.

Mod/Ref Flags: M, R

Our analysis keeps track of whether a particular memory object has been **M**odified or **R**ead within the current scope of analysis, and this is represented via the **'M'** and **'R'** flags. For example, in the `do_all` function, the statement "`L = L->Next;`" reads a pointer element from the node pointed to by `L`, which causes the **'R'** flag to be set in $flags(node(E_V(\texttt{L})))$ as shown in Figure 3.3. Mod/Ref information is useful to a variety of client analyses. Note that DSA does not track per-field mod-ref

information. While this would be an easy extension, to date it has not been needed.

Complete Information Flag: C

Practical algorithms must correctly handle incomplete programs: those where code for some functions are unavailable, or where the "program" is actually a library of code without information about the clients. In order to allow an aggressive analysis even under such situations, each DS node tracks whether there may be information missing from it.

For example, in Figure 3.3, Data Structure Analysis does not yet know anything about the incoming `L` and `FP` arguments because it hasn't performed interprocedural analysis. Inside this function, it can determine that `L` is treated as a `list` object (the construction algorithm looks at how pointers are used, not what their declared types are), that it is read from, and what nodes each variable points to. However, it can not know whether the information it has for this memory object is correct in a larger scope. For example, the `FP` and `L` arguments are speculatively represented as different objects, even though they might actually be aliased to each other when called from a particular call site.

To handle such situations, Data Structure Analysis computes which nodes in the graph are "complete," and marks each one with the **C**omplete flag[1]. If a node is not marked complete, the information calculated for the DS node represents partial information and must be treated conservatively. In particular, the node may later be assigned extra edges, extra flags, a different type, or may even end up merged with another incomplete node in the graph. For example, from the graph in Figure 3.3 an alias analysis algorithm (such as the one described in Section 4.2) must assume that `L` and `FP` may alias. Nevertheless, other nodes in such a graph may be complete and such nodes will never be merged with any other node, allowing clients to obtain useful information from graphs with partial information.

This capability is the key to the incremental nature of our algorithm: Because nodes keep track of which information is final, and which is still being created, the graphs constructed by our algorithm are *always* conservatively correct, even during intermediate steps of the analysis.

[1] This is somewhat similar to the "inside nodes" of [138].

Flag for Field-Sensitivity With and Without Type-Safety: O

A particularly important benefit of the "Complete" flag is that it allows DS Analysis to efficiently provide field-sensitive information *for the type-safe subsets of programs.* This is important because field-sensitivity for type-unsafe structure types can be very expensive [128], but in fact we observe that most portable code is completely (or mostly) type-safe, even if the source language does not require it (e.g., C or C++). The complete flag allows DS analysis to assume speculatively that all access to a node are type-safe, until an access to the node is found which conflicts with the other accesses. Because a node is not marked complete as long as there are potentially unprocessed accesses, this is safe.

DS Analysis provides field-sensitive information by associating a type, $T(n)$, with each DS node n, and keeping track of a distinct outgoing edge for each pointer field of the type. If all accesses to all objects at the node use a consistent type τ, then $T(n) = \tau$.[2]

If operations using incompatible types (as defined in Section 3.2) are found, the type for the node is treated as an unsized array of bytes ($T(n) =$ `char[]`), and the fields and edges of the node are "c**O**llapsed" into a single field with at most one outgoing edge, using the following algorithm:

collapse(dsnode n)
  cell $e = \langle$null, 0$\rangle$ // null target
  $\forall f \in fields(T(n))$
    e = mergeCells($e, E(\langle n, f \rangle)$) // merge old target with e
    remove field f // remove old edge
  $T(n) =$ `char[]` // reset type information
  $E(\langle n, 0 \rangle) = e$ // new edge from field 0
  $flags(n) = flags(n) \cup$ **'O'** // mark node C**o**llapsed

In the pseudo-code, a "cell" is a ⟨node,field⟩ pair, used as "sources" of edges in the DS graphs. The function "*mergeCells(*c_1, c_2*)*" (described in the next section) merges the cells c_1 and c_2 and therefore the nodes pointed to by those cells. This ensures that the targets of the two cells are now exactly equal. Because the above algorithm merges all outgoing edges from the node, the end result is the same as if field-sensitivity were never speculated for node n. If a node has been collapsed (i.e., $\mathbf{O} \in flags(n)$), it is always treated in this safe, but field-insensitive, manner.

[2] As Section 3.2 describes, type information is inferred only at actual accesses rather than from the declared types for variables, so that common idioms such as casting a pointer to `void*` and back do not cause a loss of precision.

3.2 Construction Algorithm

DS graphs are created and refined in a three step process. The first phase constructs a DS graph for each function in the program, using only intraprocedural information (a "local" graph). Second, a "Bottom-Up" analysis phase is used to eliminate incomplete information due to callees in a function, by incorporating information from callee graphs into the caller's graph (creating a "BU" graph). The final "Top-Down" phase eliminates incomplete information due to incoming arguments by merging caller graphs into callees (creating a "TD" graph). The BU and TD phases operate on the "known" Strongly Connected Components (SCCs) in the call graph.

Two properties are important for understanding how the analysis works in the presence of incomplete programs, and how it can incrementally construct the call graph even though it operates on the SCCs of the graph. First, the DS graph for a function is conservatively correct even if only a subset of its potential callers and potential callees have been incorporated into the graph (i.e., the information in the graph can be used safely so long as the limitations on nodes without **'C'** flags are respected, as described in Section 3.1.1). Intuitively, the key to this property simply is that a node must not be marked complete until it is known that all callers and callees potentially affecting that node have been incorporated into the graph. Second, the result of two graph inlining operations at one or two call sites is independent of the order of those operations. This follows from a more basic property that the order in which a set of nodes are merged does not affect the final result.

3.2.1 Primitive Graph Operations

Data Structure Analysis is a flow-insensitive algorithm which uses a unification-based memory model, similar to Steensgaard's algorithm [129]. The algorithm uses several primitive operations on DS graphs, shown in Figure 3.4. These operations are used in the algorithm to merge two cells, merge two nodes while aligning fields in a specified manner, to inline a callee's graph into a caller's graph at a particular call site, and vice versa. The latter two operations are described later in this section.

The fundamental operation in the algorithm is *mergeCells*, which merges the two target nodes specified. This requires merging the type information, flags, globals, outgoing edges of the two

nodes, and moving the incoming edges to the resulting node. If the two fields have incompatible types (e.g., $T(n_1) = \texttt{int}$, $f_1 = 0$, $T(n_2) = \{\texttt{int}, \texttt{short}\}$, $f_2 = 1$), or if the two node types are compatible but the fields are misaligned (e.g., $T(n_1) = T(n_2) = \{\texttt{int}, \texttt{short}\}$, $f_1 = 0$, $f_2 = 1$), the resulting node is first collapsed as described in Section 10, before the rest of the information is merged. Merging the outgoing edges causes the target node of the edges to be merged as well (if the node is collapsed, the resulting node for n_2 will have only one outgoing edge which is merged with all the out-edges of n_1). To perform this recursive merging of nodes efficiently, the merging operations are implemented using Tarjan's Union-Find algorithm.

```
(Merge two cells of same or different nodes; update n2, discard n1)
Cell mergeCells(Cell ⟨n1, f1⟩, Cell ⟨n2, f2⟩,)
  if (IncompatibleForMerge(T(n1), T(n2), f1, f2))
    collapse n2 (i.e., merge fields and out-edges)
  union flags of n1 into flags of n2
  union globals of n1 into globals of n2
  merge target of each out-edge of ⟨n1, fj⟩ with
        target of corresponding field of n2
  move in-edges of n1 to corresponding fields of n2
  destroy n1
  return ⟨n2, 0⟩ (if collapsed) or ⟨n2, f2⟩ (otherwise)

(Clone G1 into G2; merge corresponding nodes for each global)
cloneGraphInto(G1, G2)
  G1c = make a copy of graph G1
  Add nodes and edges of G1c to G2
  for (each node N ∈ G1c)
    for (each global g ∈ G(N))
      merge N with the node containing g in G2

(Clone callee graph into caller and merge arguments and return)
resolveCallee(Graph G_callee, Graph G_caller,
                              Function F_callee, CallSite CS)
  cloneGraphInto(G_callee, G_caller)
  clear 'S' flags on cloned nodes
  resolveArguments(G_caller, F_callee, CS)

(Clone caller graph into callee and merge arguments and return)
resolveCaller(Graph G_caller, Graph G_callee,
                              Function F_callee, CallSite CS)
  cloneGraphInto(G_caller, G_callee)
  resolveArguments(G_callee, F_callee, CS)

(Merge arguments and return value for resolving a call site)
resolveArguments(Graph G_merged, Function F_C, CallSite CS)
  mergeCells(target of CS[1], target of return value of F_C)
  for (1 ≤ i ≤ min(Numformals(F_C), NumActualArgs(CS))
    mergeCells(target of arg i at CS, target of formal i of F_C)
```

Figure 3.4: Primitive operations used in the algorithm

3.2.2 Local Analysis Phase

The goal of the local analysis phase is to compute a *Local DS graph* for each function, without any information about callers and callees. This is the only phase that inspects the actual program representation: the other two phases operate solely on DS graphs. All **H**,**S**,**G**,**U**,**M**,**R** flags and call nodes are derived solely in this phase: other phases only propagate them.

The local DS graph for a function F is computed as shown in Figure 3.5. We present this analysis in terms of a minimal language which is still as powerful as C. The assumptions about the type system and memory model in this language were described in Section 3.1[3].

The "*LocalAnalysis*" routine first creates an empty node as a target for every pointer-compatible virtual register (entering them in the map E_V), and creates a separate node for every global variable. The analysis then does a linear scan over the instructions of the function, creating new nodes at `malloc` and `alloca` operations, merging edges of variables at assignments and the return instruction, and updating type information at selected operations. The type of a cell, $E_V(Y)$, is updated only when Y is actually *used* in a manner that interprets its type, viz., at a dereference operation on Y (for a load or store) and when indexing into a structure or array pointed to by Y. `malloc`, `alloca`, and `cast` operations simply create a node of `void` type. Structure field accesses adjust the incoming edge to point to the addressed field (which is a no-op if the node is collapsed). Indexing into array objects is ignored, i.e., arrays are treated as having a single element. `return` instructions are handled by creating a special π virtual register which is used to capture the return value.

Function calls result in a new call node being added to the DS graph, with entries for the value returned, the function pointer (for both direct and indirect calls), and for arguments. For example, the local graph for `addGTList` in Figure 3.7(a) shows the call node created for the call to function `do_all`. Note that an empty node is created and then merged using *mergeCells* for each entry in order to correctly merge type information. This avoids losing type information when the declared type of an object is cast to an intermediate type (e.g., void*), then cast back to its declared type again.

[3]We assume that the functions $E(X)$ and $E_V(X)$ return a new, empty node with the type of X (by invoking makeNode($typeof(X)$)) when no previous edge from the cell or variable X existed. For example, in Figure 3.7(a), the incoming argument L points to such a node. We also abuse the notation by using $E(X) = \ldots$ or $E_V(X) = \ldots$ to change what X points to.

(*Compute the local DS Graph for function F*)
LocalAnalysis(function F)
  Create an empty graph
  if F returns pointer compatible type
    $E_V(\pi)$ = makeNode(void)
  $\forall$ virtual registers R, $E_V(R)$ = makeNode($T(R)$)
  $\forall$ globals X (variables and functions) used in F
    N = makeNode($T(X)$);
    $G(N) \cup = X$;
    $flags(N) \cup = \mathbf{'G'}$

  $\forall$ instruction $I \in F$: case I in:

    `X = malloc ...`: *(heap allocation)*
      $E_V(X) = makeNode($`void`$)$
      $flags(node(E_V(X))) \cup =$ **'H'**

    `X = alloca ...`: *(stack allocation)*
      $E_V(X) = makeNode($`void`$)$
      $flags(node(E_V(X))) \cup =$ **'S'**

    `X = *Y`:
      mergeCells($E_V(X)$, $E(E_V(Y))$)
      $flags(node(E_V(X)) \cup =$ **'R'**

    `*Y = X`:
      mergeCells($E_V(X)$, $E(E_V(Y))$)
      $flags(node(E_V(X)) \cup =$ **'M'**

    `X = &Y->Z`: *(address of struct field)*
      $\langle n, f \rangle$ = updateType($E_V(Y)$, $typeof(*Y)$)
      $f' = 0$, if n is collapsed; $field(field(n, f), Z)$ otherwise
      mergeCells($E_V(X), \langle n, f' \rangle$)

    `X = &Y[idx]`: *(address of array element)*
      $\langle n, f \rangle$ = updateType($E_V(Y)$, $typeof(*Y)$)
      mergeCells($E_V(X), \langle n, f \rangle$)

    `return X`: *(return pointer-compatible value)*
      mergeCells($E_V(\pi), E_V(X)$)

    `X = (`τ`) Y`: *(value-preserving cast)*
      mergeCells($E_V(X), E_V(Y)$)

    `X = Y(Z`$_1$`, Z`$_2$`, ... Z`$_n$`)`: *(function call)*
      callnode c = new callnode
      $C \cup = c$
      mergeCells($E_V(X), c[1]$)
      mergeCells($E_V(Y), c[2]$)
      $\forall i \in \{1...n\}$: mergeCells($E_V(Z_i), c[i+2]$)

    (Otherwise) `X = Y op Z`: *(all other instructions)*

      mergeCells($E_V(X), E_V(Y)$)
      mergeCells($E_V(X), E_V(Z)$)
      $flags(node(E_V(X))) \cup =$ **'U'**
      collapse($node(E_V(X))$)

  MarkCompleteNodes()

Figure 3.5: The LocalAnalysis function

(*Create a new, empty node of type* τ)
makeNode(type τ)
  n = new Node(type = τ, flags = ϕ, globals = ϕ)
  $\forall\ f\ \in\ fields(\tau)$, $E(n, f) =< null, 0 >$
  return n

(*Merge type of field* $\langle n, f\rangle$ *with type* τ. This may
collapse fields and update in/out edges via mergeCells())
updateType(cell $\langle n, f\rangle$, type τ)
  if ($\tau \neq$ `void` $\wedge\ \tau\ \neq\ typeof(\langle n, f\rangle)$)
    m = makeNode(τ)
    return mergeCells($\langle m, 0\rangle, \langle n, f\rangle$)
  else return $\langle n, f\rangle$

Figure 3.6: makeNode and updateType operations

Finally, if any other instruction is applied to a pointer-compatible value, (e.g., a cast from a pointer to an integer smaller than the pointer, or integer arithmetic), any nodes pointed to by operands and the result are collapsed and the **U**nknown flag is set on the node[4].

The final step in local graph construction is to calculate which DS nodes are **C**omplete. For a Local graph, nodes reachable from a formal argument, a global, passed as an argument to a call site, or returned by a function call may not be marked complete. This reflects the fact that the local analysis phase does not have any interprocedural information. For example, in Figure 3.7(a), neither of the nodes for for the arguments to `do_all` are marked **'C'**.

3.2.3 Bottom-Up Analysis Phase

The Bottom-Up (BU) analysis phase refines the local graph for each function by incorporating interprocedural information from the callees of each function. The result of the BU analysis is a graph for each function which summarizes the total effect of calling that function (e.g., the imposed aliases and mod/ref information) without any calling context information. It computes this graph by cloning the BU graphs of all *known* callees into the caller's Local graph, merging nodes pointed to by corresponding formal and actual arguments.

The Bottom-Up analysis is the key pass involved in computing the fully context-sensitive analysis result by cloning and inlining graphs from callees into callers. Cloning graphs for each edge in the call graph directly provides a fully context sensitive result by implicitly[5] naming objects by the

[4]In LLVM, type-safe pointer arithmetic is represented with the `getelementptr` operation, which effectively computes `&Y->Z` or `&Y[idx]`. See Section 2.2.2.

[5]The naming is implicit because we do not explicitly remember where a node was inlined from. This is one of the key ways we maintain aggressive scalability.

call path they are inlined from. Cloning nodes is an inherently exponential process, but is controlled by three factors: 1) unification merges most cloned nodes together (e.g., often summarizing lists as recursive nodes) 2) memory objects that are unreachable in a caller are not copied from a callee, and 3) Nodes corresponding to global variables are always merged as inlining occurs (i.e., the node for a global G in a callee is merged with the node for G in the caller if it exists), which leads to recursive merging due to #1. In practice, while exponential behavior is theoretically possible, we find that it does not occur in practice, Section 3.2.6 describes how to handle it if it does happen.

We first describe a single graph inlining operation, and then explain how the call graph is discovered and traversed. Consider a call to a function F with formal arguments $f_1,\ldots,f_n$, where the actual arguments passed are $a_1,\ldots,a_n$. The function *resolveCallee* in Figure 3.4 shows how such a call is processed in the BU phase. We first copy the BU graph for F, clearing all **S**tack node markers since stack objects of a callee are not legally accessible in a caller. We then merge the node pointed to by each actual argument a_i of pointer-compatible type with the copy of the node pointed to by f_i. If applicable, we also merge the return value in the call node with the copy of the return value node from the callee. Note that any unresolved call nodes in F's BU graph are copied into the caller's graph, and all the objects representing arguments of the unresolved call in the callee's graph are now represented in the caller as well.

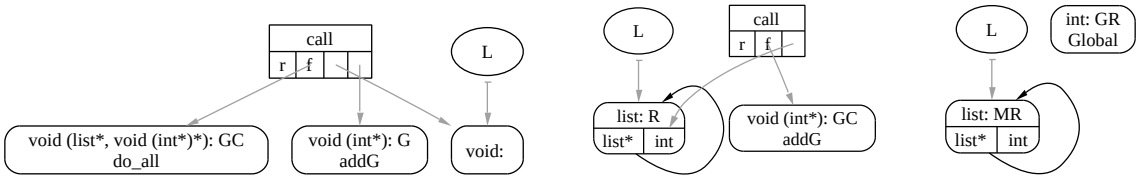
Basic Analysis Without Recursion

The complete Bottom-Up algorithm for traversing the call graph is shown in Figure 3.8. but we explain it for four different cases. In the simplest case of a program with only direct calls to non-external functions, no recursion, and no function pointers, the call nodes in each DS graph implicitly define the entire call graph. The BU phase simply has to traverse this acyclic call graph in post-order (visiting callees before callers), cloning and inlining graphs as described above.

To support programs that have function pointers and external functions (but no recursion), we simply restrict our post-order traversal to only process a call-site if its function pointer targets a **C**omplete node (i.e., its targets are are fully resolved, as explained in Section 3.1.1), *and* all potential callees are non-external functions (line 1 in the Figure).

Such a call site may become resolved if the function passed to a function pointer argument

becomes known. For example, the call to *FP* cannot be resolved within the function `do_all`, but will be resolved in the BU graph for the function `addGToList`, where we conclude that it is a call to *addG*. We clone and merge the indirect callee's BU graph into the graph of the function where the call site became resolved, merging actual and formal arguments as well as return values, using *resolveCallee* just as before (line 2 in the figure). This technique of resolving call nodes as their function pointer targets are completed effectively discovers the call-graph on the fly, and we record the call graph as it is discovered.



(a) Local `addGToList` graph (b) After inlining `do_all` (c) Final BU graph

Figure 3.7: Construction of the BU DS graph for `addGToList`

Note that the BU graph of the function containing the original call still has the unresolved call node. We do not re-visit previously visited functions in each phase, but that call node will eventually be resolved in the top-down phase. The BU graph for the function where the call was resolved now fully incorporates the effect of the call. For example, inlining the BU graph of `addG` into that of `addGToList` yields the finished graph shown in Figure 3.7(c). The **M**odified flag in the node pointed to by L is obtained from the node $E_V(X)$ from `addG` (Figure 3.3), which is merged with the second argument node inlined from `do_all`. This graph for `addGToList` is identical to that which would have been obtained if `addG` was first inlined into `do_all` (eliminating the call node) and the resulting graph was then inlined into `addGToList`.

After the cloning and merging is complete for a function in the SCC, we identify new complete nodes (Section 3.2.2) (line 5) and remove unreachable nodes from the graph (line 6). The latter are created because copying and inlining callee graphs can bring in excess nodes not accessible within the current function (and therefore not accessible in any of its callers as well). This includes non-global nodes not reachable from any virtual register, global node, or call node.

Recursion without Function Pointers

Our strategy for handling recursion is essentially to apply the bottom-up process described above but on Strongly Connected Components (SCCs) of the call graph, handling each multi-node SCC separately. The key difficulty is that call edges are not known beforehand and, instead, are discovered incrementally by the algorithm (implying that cycles are incrementally discovered as well). The overall Bottom-Up analysis algorithm is shown in Figure 3.8. It uses an adaptation of Tarjan's linear-time algorithm to find and visit Strongly Connected Components (SCCs) in the call graph in postorder [118].

Assume first that there are only direct calls, i.e., the call graph is known. For each SCC, all calls to functions outside the SCC are first cloned and resolved as before (these functions will already have been visited because of the postorder traversal over SCCs). Once this step is complete, all of the functions in the SCC have empty lists of call sites, except for intra-SCC calls and calls to external functions (the latter are simply ignored throughout). In an SCC, each function will eventually need to inline the graphs of all other functions in the SCC at least once (either directly or through the graph of a callee). A naive algorithm can produce an exponential number of inlining operations, and even a careful enumeration can require $O(n^2)$ inlining operations in complex SCCs (which we encountered in some programs).

Instead, because there are an infinite number of call paths through the SCC, we choose to completely ignore intra-SCC context-sensitivity. We merge the partial BU graphs of all functions in the SCC, resolving all intra-SCC calls in the context of this single merged graph, capturing the same information as other fully context-sensitive algorithms [140]. A more aggressive technique would try to preserve some of the context-sensitivity within an SCC for better precision, but we found this approach to be inscalable (thus we leave it to future work).

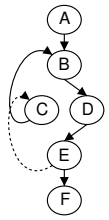
Recursion with Function Pointers

The final case to consider is a recursive program with indirect calls. The difficulty is that some indirect calls may induce cycles in the SCC, but these call edges will not be discovered until the indirect call is resolved. We make a key observation, based on the properties described earlier, that yields a simple strategy to handle such situations: some call edges of an SCC can be resolved *before*

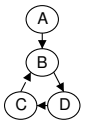
```
BottomUpAnalysis(Program P)
  ∀ Function F ∈ P
.   BUGraph{F} = LocalGraph{F}
.   Val[F] = 0; NextID = 0
  while (∃ unvisited functions F ∈ P)    (visit main first if available)
    TarjanVisitNode(F, new Stack)

TarjanVisitNode(Function F, Stack Stk)
  NextID++; Val[F] = NextID; MinVisit = NextID; Stk.push(F)
  ∀ call sites C ∈ BUGraph{F}
    ∀ known non-external callees F_C at C
      if (Val[F_C] == 0)        (F_C unvisited)
        TarjanVisitNode(F_C, S)
      else MinVisit = min(MinVisit, Val[F_C])
    if (MinVisit == Val[F])      (new SCC at top of Stack)
      SCC S = { N: N = F ∨ N appears above F on stack }
      ∀ F ∈ S: Val[F] = MAXINT; Stk.pop(F)
      ProcessSCC(S, Stk)

ProcessSCC(SCC S, Stack Stk)
  ∀ Function F ∈ S
(1)   ∀ resolvable call sites C ∈ BUGraph{F}       (see text)
        ∀ known callees F_C at C
          if (F_C ∉ S)        (Process funcs not in SCC)
(2)         ResolveCallee(BUGraph{F_C}, BUGraph{F}, F_C, CS)
(3)   SCCGraph = BUGraph{F_0}, for some F_0 ∈ S
      ∀ Function F ∈ S, F ≠ F_0    (Merge all BUGraphs of SCC)
        cloneGraphInto(BUGraph{F}, SCCGraph)
        BUGraph{F} = SCCGraph
(4)   ∀ resolvable call sites C ∈ SCCGraph        (see text)
        ∀ known callees F_C at C    (Note: F_C ∈ S)
          ResolveArguments(SCCGraph, F_C, CS)
(5)   MarkCompleteNodes() - Section 3.2.2
(6)   remove unreachable nodes
(7)   if (SCCGraph contains new resolvable call sites)
        ∀ F ∈ S: Val[F] = 0      (mark unvisited)
        TarjanVisitNode(F_0, Stk), for some F_0 ∈ S   (Re-visit SCC)
```

Figure 3.8: Bottom-Up Closure Algorithm

discovering that they form part of an SCC. When the call site "closing the cycle" is discovered (say in the context of a function F_0), the effect of the complete SCC will be incorporated into the BU graph for F_0 though not the graphs for functions handled earlier.



(a) Recursive Call Graph (indirect call is dotted)



(b) Call Node Edges, After inlining F & E

1. { F }
2. { E }
3. { D }: mark unvisited
4. { B, D, C }
5. { A }

(c) SCC visitation order

Figure 3.9: Handling recursion due to an indirect call in the Bottom-Up phase

Based on this observation, we slightly adapted Tarjan's algorithm to revisit nodes of an SCC when the SCC is discovered, even though some of the nodes may have been visited earlier (but visiting only unresolved call sites). After the current SCC is fully processed (i.e., after step (5) in Figure 3.8), we check whether the SCC graph contains any newly inlined call nodes that are now resolvable. If so, we reset the *Val* entries for all functions in the SCC, which are used in *TarjanVisitNode* to check if a node has been visited. the nodes in the *current* SCC to be revisited, but only the new call sites are processed (since other resolvable call sites have already been resolved, and will not be included in steps (1) and (4)). Note that this is a simple form of a partially dynamic incremental online SCC finding algorithm [104].

For example, consider the recursive call graph shown in Figure 3.9(a), where the call from E to C is an indirect call. Assume this call is resolved in function D, e.g., because D passes C explicitly to E as a function pointer argument. Since the edge $E \rightarrow C$ is unknown when visiting E, Tarjan's algorithm will first discover the SCCs { F }, { E }, and then { D } (Figure 3.9(c)). Now, it will find a new call node in the graph for D, find it is resolvable as a call to C, and mark D as unvisited (Figure 3.9(b)). This causes Tarjan's algorithm to visit the "phantom" edge $D \rightarrow C$, and therefore to discover the partial SCC { B, D, C }. After processing this SCC, no new call nodes are discovered. At this point, the BU graphs for B, D and C will all correctly reflect the effect of the call from E to C, but the graph for E will not[6]. The top-down pass will resolve the call from

[6]Nor should it. A different caller of E may cause the edge to be resolved to a different function, thus the BU graph for E does not include information about a call edge which is not necessarily present in all calling contexts.

E to C (within E) by inlining the graph for D into E.

Note that even in this case, the algorithm only resolves each callee at each call site once: no iteration is required, even for SCCs induced by indirect calls.

The graph of Figure 3.10 shows the BU graph calculated for the `main` function of our example. This graph has disjoint subgraphs for the lists pointed to by X and Y. These were proved disjoint because we cloned and then inlined the BU graph for each call to `addGToList()`. This shows how the combination of context sensitivity with cloning can identify disjoint data structures, even when complex pointer manipulation is involved.

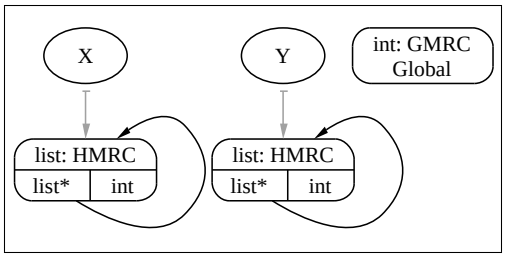


Figure 3.10: Finished BU graph for `main`

3.2.4 Top-Down Analysis Phase

The Top-Down construction phase is very similar to the Bottom-Up construction phase. The BU phase has already identified the call graph, so the TD phase can traverse the SCCs of the call graph directly using Tarjan's algorithm; it does not need to "re-visit" SCCs as the BU phase does. Note that some SCCs may have been visited only partially in the BU phase, so the TD phase is responsible for merging their graphs.

Overall, the TD phase differs from the BU phase in only 4 ways: First, the TD phase never marks an SCC as unvisited as explained above: it uses the call edges discovered and recorded by the BU phase. Second, the TD phase visits SCCs of the call graph computed by the Bottom-Up traversal in reverse postorder instead of postorder. Third, the Top-Down pass inlines each function's graph into each of its callees (rather than the reverse), and it inlines a caller's graph into all it's potential callees directly (it never needs to "defer" this inlining operation since the potential callees at each call site are known). The final difference is that formal argument nodes are marked complete if all callers of a function have been identified by the analysis, i.e., the function is not

accessible to any external functions. Similarly, global variables may be marked complete, unless they are accessible to external functions. A function or global escapes the program if it does not have internal linkage (i.e., it is not marked static in C) or if the exists a node for the global in `main`'s graph that is not marked **C**omplete.

3.2.5 Complexity Analysis

The local phase adds at most one new node, ScalarMap entry, and/or edge for each instruction in a procedure (before node merging). Furthermore, node merging or collapsing only reduces the number of nodes and edges in the graphs. We implemented node merging using a Union-Find data structure, which ensures that the local phase requires $O(n\alpha(n))$ time and $O(n)$ space for a program containing n instructions in all [129].

The BU and TD phases operate on DS graphs directly, so their performance depends on the size of the graphs being cloned and the time to clone and merge one graph into another. We denote these by K and T_{inline} respectively, where T_{inline} is $O(K\alpha(K))$ in the worst case. They also depend on the average number of callee functions per caller (not call site), denoted c.

For the BU phase, each function must inline the graphs for c callee functions, on average. Because each inlining operation requires T_{inline} time, this requires fcT_{inline} time if there are f functions in the program. The call sites within an SCC do not introduce additional complexity, since every potential callee is again inlined only once into its caller within or outside the SCC (in fact, these are slightly faster because only a single graph is built, causing common nodes to be merged). Thus, the time to compute the BU graph is $\Theta(fcT_{inline})$. The space required to represent the Bottom-Up graphs is $\Theta(fK)$. The TD phase is identical in complexity to the BU phase.

3.2.6 Bounding Graph Size

In the common case, the merging behavior of the unification algorithm we use keeps individual data structure graphs very compact, which occurs whenever a data structure is processed by a loop or recursion. Nevertheless, the combination of field sensitivity and cloning makes it theoretically possible for a program to build data structure graphs that are exponential in the size of the input program. Such cases can only occur if the program builds and processes a large complex data

structure using only non-loop, non-recursive code, and are thus *extremely* unlikely to occur in practice.

Using a technique like k-limiting [73] to guard against such unlikely cases is unattractive because it could reduce precision for reasonable data structures with paths more than k nodes long. Instead, we propose that implementations simply impose a hard limit on graph size (10,000 nodes, for example, which is much larger than any real program is likely to need). If this limit is exceeded, node merging can be used to reduce the size of the graph. Because this is only a theoretical concern, our implementation does not include the check. Our results in Section 3.4 show that the maximum function graph size we observed in practice across a wide range of programs is quite small.

3.3 Engineering an Efficient Pointer Analysis

As part of its basic design, DSA includes several features that are required for scalability. For example, the use of unification solves the exponential explosion inherent in cloning in practice. Additionally, processing SCC's in the call graph eliminates the need for iteration inside of SCC's. Other factors are less obvious. In particular, because the local phase is the only part of DSA that uses the compiler IR (all other phases perform graph transformations on DS Graphs), DSA has better cache behavior than analyses that need to keep the pointer representation *and* the compiler IR in cache.

These design choices are some of the keys to achieving practical analyses, and can reduce analysis times by several orders of magnitude. In addition to these key design choices, this section lists several important engineering issues which can also improve analysis times in important cases, primarily by improving handling of global variables and by reducing N^2 behavior in important cases.

3.3.1 The Globals Graph

One reason the DS graph representation is so compact is that each function graph need only contain the memory reachable from that function. However, Figures 3.7(c) and 3.10 illustrate a fundamental violation of this strength. In both of these graphs, the global variable `G` makes an appearance even though it is not directly referenced and no edges target it. Such nodes cannot simply be deleted

because they may have to be merged with other nodes in callers or callees of each function. If left untreated, all global variables defined in the program would propagate bottom-up to `main`, then top-down to all functions in the program. This trivially balloons the size of each graph to include every global variable in the program, a potential $O(N^2)$ size explosion.

In order to prevent this unacceptable behavior, our implementation uses a separate "Globals Graph" to hold information about global nodes and all nodes reachable from global nodes. This allows us to remove global variables from a function's graph if they are not used in the current function (even though they may be used in callers or callees of that function). For example, this eliminates the two `G` nodes in the example graphs[7].

For the steps below, all nodes reachable from virtual registers (which includes formal parameters and return values of the current function, and call node arguments within the current function, but not globals) are considered to be *locally used*. Call nodes are also considered to be locally used, unless they contain a callee that is an external function (and thus will never be resolved).

More specifically, we make the following changes to the algorithm:

- In the BU phase (respectively, TD phase), after all known callees (respectively, callers) have been incorporated in step 4, we copy and merge in the nodes from the globals graph for every global G that has a node in the current graph, plus any nodes reachable from such nodes. This ensures that the current graph reflects all known information about such globals from other functions.

- After step 5 in the BU phase, we copy all global nodes and nodes reachable from such nodes into the globals graph, merging the global nodes with the corresponding nodes already in the Globals Graph, if any (which will cause other "corresponding" nodes to be merged as well). We clear the **S**tack markers on nodes being copied into the Globals Graph, for the same reason as in *ResolveCallee*. We also clear the **C**omplete markers since those markers will be re-computed correctly within the context of each function.

  By the end of the BU phase, all the known behavior about globals will be reflected in the Globals Graph. Therefore, globals do not need to be copied from the TD graph to the Globals graph in the TD phase.

[7] Liang and Harrold [92] use a somewhat similar technique.

- In step 6 of the BU phase, we identify global nodes that are not reachable from any locally used nodes *and do not reach any such nodes*. The latter requirement is necessary because we may revisit the current function later, resolving previously unresolved call sites, which can bring in additional globals. Merging such globals will not correctly merge other reachable nodes in the graph if a global that can reach a locally reachable node is removed from the graph. The latter requirement is not needed for the TD phase since no further inlining needs to happen after reaching step 6. We simply drop all these identified nodes from the BU or TD graph for the function.

In practice, we find that the Globals graph to make a remarkable difference in running time for global-intensive programs, speeding up the top-down phase by an order of magnitude or more.

3.3.2 Efficient Graph Inlining

Our first implementation of DSA used a very simple implementation of the graph inlining operation described in Section 3.2.1. To inline a callee graph into a caller graph (for example), it literally made a copy of the callee graph into the caller graph, then used unification to perform the merge (this algorithm is listed as the `cloneGraphInto` operation in Figure 3.4). The merge simply unifies each of the linked nodes between the caller and callee: this includes the formal/actual argument bindings as well as any global variables that are common to the two graphs.

This implementation is inefficient for several reasons. First, this operation copies nodes that are not reachable in the caller graph (e.g. for stack allocations in the callee or local data structures), requiring an "unreachable node elimination" cleanup pass to get rid of them. Second, copying nodes only to unify them away is a gross waste of time. Third, unification uses a union-find approach which does not immediately free a node when it is unified. In particular, all nodes referring to a unified node need to have their references updated (lazily), which means the nodes that are copied may last far longer than we would like (consuming memory).

To solve these problems, our implementation uses a parallel recursive traversal of the caller and callee graphs starting from each matching pair of callee and caller nodes. For each pair of nodes traversed, we merge information from the callee node into the caller node (which may involve merging or collapsing nodes in the caller graph). If no caller node corresponds to the callee node,

nodes are lazily (recursively) created. Nodes that exist in the caller but not the callee do not require recursive traversal.

This approach solves all of the problems with the naive implementation: 1) only reachable nodes are copied. 2) the only new nodes created are those that exist in the callee graph but not in the caller graph. 3) The dead nodes are never created, so they do not use memory or time.

3.3.3 Partitioning E_V for Efficient Global Variable Iteration

The E_V mapping described in Section 3.1 contains all of the scalar pointers in the graph as well as the addresses of all globals. This mapping is used primarily by clients of the analysis (e.g. to find out which node a pointer points to), but is also used by various phases of the analysis (e.g. to find the formal arguments for a function when inlining a graph). In programs with large SCCs (and thus many functions merged into the same DS graph), this mapping can be very large.

Several portions of the DSA algorithm need access to all of the global variables that exist in a DSGraph (e.g. updating the globals graph, and performing graph inlining operations). Our initial implementation iterated through the E_V to find the globals used in a graph, which suffered due to the large size of E_V (while clients use constant-time hash-table lookups, iteration takes linear time).

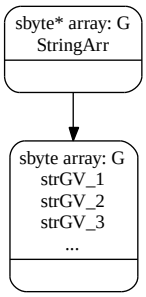
Our solution is to partition E_V into two mappings, one for scalar pointers and one to represent the address of globals. This allows direct iteration over just the information needed, yielding a large speedup on big codes with large call graph SCCs or many pointer variables.

3.3.4 Shrinking E_V with Global Value Equivalence Classes

Even with the refinements described in Section 3.3.1 and Section 3.3.3, program that use *extremely* large tables of global variable pointers can cause a problem. In particular, consider a program that contains the (very reasonable and not uncommon) C code shown in Figure 3.11. The figure also shows the LLVM code it expands into.

At the LLVM level, each constant string is lowered to a different global variable which is initialized with the string constant, "strGV_n" in our example (See Section 7). The "`StringArr`" global is an array that points to all of these globals, and DSA will represent this configuration with

```
const char * const StringArr[] = {
   "string1",
   "string2",
   "string3",
   ...
};
```




```
%strGV_1 = internal constant [8 x sbyte] c"string1\00"
%strGV_2 = internal constant [8 x sbyte] c"string2\00"
%strGV_3 = internal constant [8 x sbyte] c"string3\00"
%StringArr = constant [3 x sbyte*] [
   sbyte* getelementptr ([8 x sbyte]* %strGV_1, int 0, int 0),
   sbyte* getelementptr ([8 x sbyte]* %strGV_2, int 0, int 0),
   sbyte* getelementptr ([8 x sbyte]* %strGV_3, int 0, int 0)
]
```

Figure 3.11: C Source, DSGraph, and LLVM code for Global Value Equivalence Class Example

the graph shown on the right side of Figure 3.11.

Given the operation of the Globals Graph, many functions that either directly or indirectly use StringArr will have a copy of this graph in their per-function graphs. Unfortunately, this means that each of those graphs must also have E_V entries for each of the (potentially thousands) globals that are merged into the string constant node. These extra entries slow down any analyses that need to iterate over globals in the graph and require extra memory to represent. Finally, note that DSA will *never* be able to distinguish between the `strGV_*` nodes in the graph.

The solution we use for this problem is to maintain an equivalence class of global value addresses, merging these equivalence classes (maintained with Tarjan's union-find algorithm) when DSA merges nodes corresponding to multiple globals. With this refinement, DSA need only keep the leader of an equivalence class in the graphs. The interface used to query the DSGraphs automatically return the full set of globals in the equivalence class, permitting clients to be unaware of this implementation detail. In practice, we find that this straight-forward refinement can cut DSA runtimes by a 30% and reduce memory usage by 50% for large programs (such as 176.gcc and 253.perlbmk).

3.3.5 Avoiding N^2 Inlining for Function Pointers

With a straight-forward implementation, large tables of function pointers cause cause an efficiency problem for both the bottom-up and top-down analysis phases. The problem is that any call through the table can reach N callees, and programs with tables often have a large number of calls through them. Because of this, the BU and TD passes have to inline all N graphs M times (one for each call through the table), which takes $N * M$ time. In practice, this time can be unacceptably large for programs with hundreds of function pointers in a table.

Our solution to this problem is to keep a graph cache of all sets of function pointers inlined. For example, in the BU phase, every time a call site with more than one callee needs to be inlined, the cache is queried. If there is no entry for this set of callees, a new DSGraph is allocated, all of the callee graphs are inlined into it, all formals and globals are merged, and the new graph is added to the cache. Finally, whether the graph was in the cache or not, the graph (which now represents the effects of all callees) is inlined into the caller graph. This makes the first inline operation for a set of callees slightly more expensive for the benefit of subsequent inline operations with the same set of callees.

In the best case, instead of performing $N * M$ graph inlining operations, the BU-pass now needs to perform $N + M + 1$ graph inlining operations, a substantial improvement. In the worst case, entries in the cache are never reused, which adds one extra graph inline operation to a call site with many callees. In practice, this refinement is extremely important for certain classes of large programs.

3.3.6 Merge Call Nodes for External Functions

One simple observation is that any nodes reachable from an unrecognized external function call will always be marked incomplete. Because of this, no DSA client will be able to do any substantial analysis or transformation of these nodes. There are several ways to use this to shrink graphs: the compiler could simply merge all nodes reachable from any external function call.

For our implementation, we considered this too drastic: it eliminates the possibility of performing modular analysis (e.g. analyze a library, generate DS Graphs for it, then use these precomputed graphs when compiling the main application). As a compromise, our implementation merges call

nodes for external calls to the same function: this discards some amount of context sensitivity, but does not grossly pessimize the points-to information for external function calls[8].

In practice, we find that this can greatly reduce the number of nodes to common functions like `printf`, which often have globals (constant strings) passed as arguments. With this refinement, there is at most one node for printf format strings (per function), which contains all of the format strings in that context.

3.3.7 Direct Call Nodes

The final, and most simple, refinement is based on the observation that direct function calls are far more common than indirect function calls. As such, our representation of call nodes allows either a callee *node* (as described above) or a callee *function* to be specified for the call. In the case of direct function calls, this eliminates the need to allocate a DSNode to represent the callee of direct calls. In the case of indirect calls, a node is used to allow lazy resolution and multiple callees to be represented.

3.4 Experimental Results

We implemented the complete Data Structure Analysis algorithm in the LLVM Compiler Infrastructure (Chapter 2). The analysis is performed entirely at link-time, using stubs for standard C library functions to reflect their behavior (as in other work, e.g., [25]). To evaluate the effectiveness of DSA, we are primarily interested in four things: 1) is it fast and scalable enough for use in a commercial compiler? 2) Is the analysis memory consumption reasonable? 3) How much type information is DSA able to infer from programs? 4) How precise is DSA for alias analysis?

This section addresses the first three questions, and Chapter 4 addresses the fourth.

3.4.1 Benchmark Suite and Simple Measurements

We evaluated DSA on three benchmark suites: SPEC CPU 95, SPEC CPU 2000, and a collection of unbundled programs (which includes Povray 3.1, NAMD, boxed-sim and fpgrowth). In the SPEC

[8] In particular if a function call is passed two pointers, the nodes corresponding to these pointers would not be merged.

suites, we included all programs written in C or C++ as well as those FORTRAN 77 programs that could be converted to C by Version 20031025 of the "f2c" program. We include Povray, NAMD and boxed-sim because they have been used in other pointer analysis papers and fpgrowth is used in Chapter 5. These program range from 190 to 222,208 raw lines of source code.

Before analysis, each of these programs are compiled and linked by LLVM, being subjected to the standard suite of compile- and link-time optimizations. As part of linking C++ and FORTRAN programs, we statically link the standard runtime library into the program (libstdc++ or libf2c) as LLVM code. While LLVM includes an aggressive link-time interprocedural optimizer (which performs inlining, dead argument elimination, interprocedural constant propagation, dead global elimination, etc), it does not include any aggressive interprocedural pointer analysis.

Figure 3.12 captures some of the key properties of the benchmarks we are considering, seperated by benchmark suite. The first set of columns are indicators of static benchmark size. The "Raw LOC" column is the number of source lines of code, as counted by "wc -l". Because raw lines of code are not a very reliable metric (it includes comments, is affected by number of header files, changes impact based on source language, does not include the statically linked standard library, etc), we include a count of the number of memory instructions[9] in the analyzed LLVM code for the program. Because DSA ignores all non-memory instructions, this gives a much more reliable way to gauge the relative sizes of programs. The "max —SCC—" column shows the size of the largest SCC in the call graph for the program, as determined by DSA. Several of the programs in this collection have large call graph SCCs (for example, 176.gcc, 253.perlbmk, and povray).

The second set of columns capture information about the final Top-Down graphs computed by DSA. The first column is the total number of nodes in all Top-Down graphs, the second column is the total number of collapsed nodes in all graphs. The third column is maximum number of nodes in any Top-Down graph, and the final column is the size of the globals graph computed for the program (as described in Section 3.3.1).

These statistics show that exponential graph explosion simply doesn't happen for DSA, as mentioned in Section 3.2.6. Though DSA uses full context-sensitive cloning (and is thus susceptible to exponential behavior in theory), the unification approach used effectively eliminates this in two

[9]Memory instructions are `load`, `store`, `malloc`, `alloca`, `call`, `invoke`, and `getelementptr` instructions.

Benchmark	Code Size: Raw LOC	Code Size: Memory Instrs	Code Size: max |SCC|	TD Graph Info: Total Nodes	TD Graph Info: Collapsed Nodes	TD Graph Info: Max Nodes in a Graph	TD Graph Info: Globals Graph Size
SPEC CINT 2000							
181.mcf	2412	991	1	103	0	49	39
256.bzip2	4647	1315	1	205	3	76	85
164.gzip	8616	1785	1	290	1	60	120
175.vpr	17728	8972	1	2106	118	366	677
197.parser	11391	10086	3	1291	121	109	487
186.crafty	20650	14035	2	2890	45	701	1211
300.twolf	20459	19686	1	2022	37	411	645
255.vortex	67220	37601	23	3515	241	392	967
254.gap	71363	47389	9	5889	728	370	772
252.eon	35819	51897	6	6936	511	411	419
253.perlbmk	85055	98386	250	2038	510	401	547
176.gcc	222208	139790	337	12736	1000	3196	2876
SPEC CFP 2000							
179.art	1283	773	1	166	0	55	74
183.equake	1513	1340	1	204	0	118	86
171.swim	435	3716	2	425	16	40	123
172.mgrid	489	4064	2	530	31	40	148
168.wupwise	2184	5087	2	608	33	64	213
173.applu	3980	5966	2	593	19	68	249
188.ammp	13483	10551	1	897	69	281	316
177.mesa	58724	43352	1	3038	857	98	518
SPEC CINT 1995							
129.compress	1934	326	1	75	2	18	42
130.li	7598	7894	24	806	328	33	154
124.m88ksim	19233	7951	2	1796	195	56	571
132.ijpeg	28178	12507	1	1531	62	65	173
099.go	29246	20543	1	2298	0	131	269
134.perl	26870	29940	19	1463	136	232	553
147.vortex	67211	37632	23	3529	242	355	970
126.gcc	205085	129083	255	12226	1109	3046	2564
SPEC CFP 1995							
102.swim	429	3493	2	427	15	40	132
101.tomcatv	190	3797	2	512	19	40	153
107.mgrid	484	4010	2	519	31	40	144
145.fpppp	2784	4447	2	623	43	48	314
104.hydro2d	4292	5773	2	688	88	48	200
110.applu	3868	5854	2	583	19	57	250
103.su2cor	2332	6450	2	1080	49	160	411
146.wave5	7764	11333	2	1171	164	70	538
Other Programs							
fpgrowth	634	544	1	108	0	49	29
boxed-sim	11641	12287	1	480	61	65	151
NAMD	5312	19002	1	1539	276	224	196
povray31	108273	62734	56	5278	732	318	1044

Figure 3.12: Benchmark Suite and Basic DSA Measurements

ways: 1) unifications inherently merges together most of the nodes created through the cloning process. 2) In the case of analysis failure, when the analysis must assume that many nodes must all point to each other, unification based approaches aggressively merge these nodes, shrinking the representation (e.g., 253.perlbmk, which is largely not type-safe).

3.4.2 Analysis Time & Memory Consumption

We evaluated the time and space usage of our analysis on a Linux workstation with an AMD Athlon MP 2100+ processor. We compiled LLVM with GCC 3.4.2 at the `-O3` level of optimization. Figure 3.13 and 3.14 show the analysis time and memory usage[10] of DSA, compared against the number of LLVM memory instructions in the program, for each of the programs listed in Figure 3.12, and Figure 3.15 lists the raw data. The graphs show that DSA is both *extremely fast* and *extremely space efficient*, requiring less than **3.5s** and **20MB of memory** to fully analyze the largest program (176.gcc, which consists of 222K lines of C code). Note that memory consumption, not time, is often one of the biggest bottlenecks for interprocedural analysis: DSA has a very small footprint compared to many pointer analyses[11].

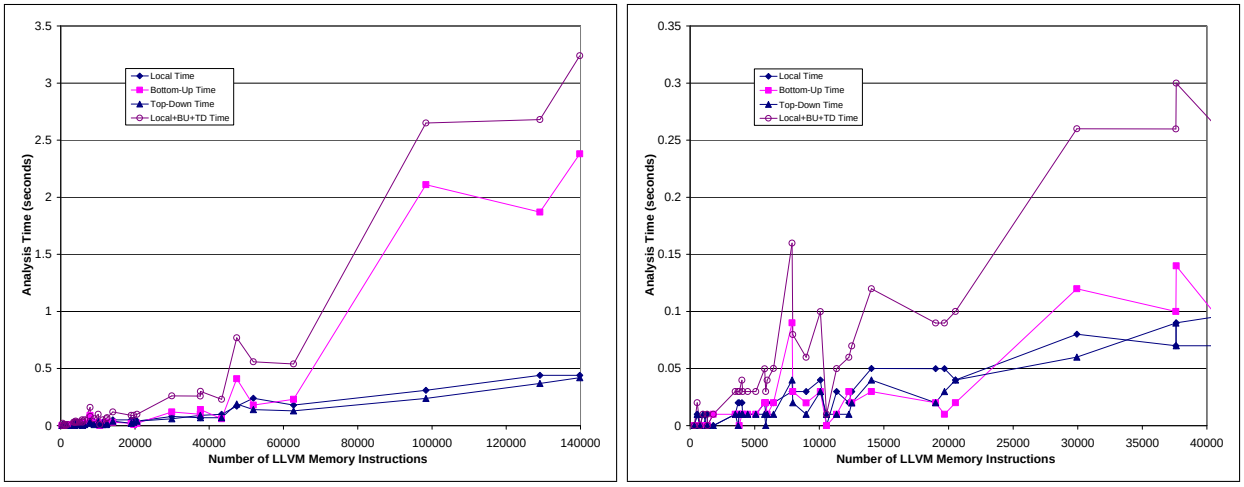


Figure 3.13: Scaling of Analysis Time with Program Size (Number of Memory Operations)
Left chart includes full data set. Right chart is zoomed in on lower-left quadrant.

[10]Note that the persistent memory footprint of the DSA results are the BU+TD sizes, as our implementation of the BU pass modifies the Local graphs in place as it is computed (the local graphs are not useful to any clients, so they do not need to be preserved). See Section 4.2 for details.

[11]Even in the closest comparable analysis [92], for example, field-sensitivity had to be disabled for the `povray3` program for the analysis to fit into 640M of physical memory.

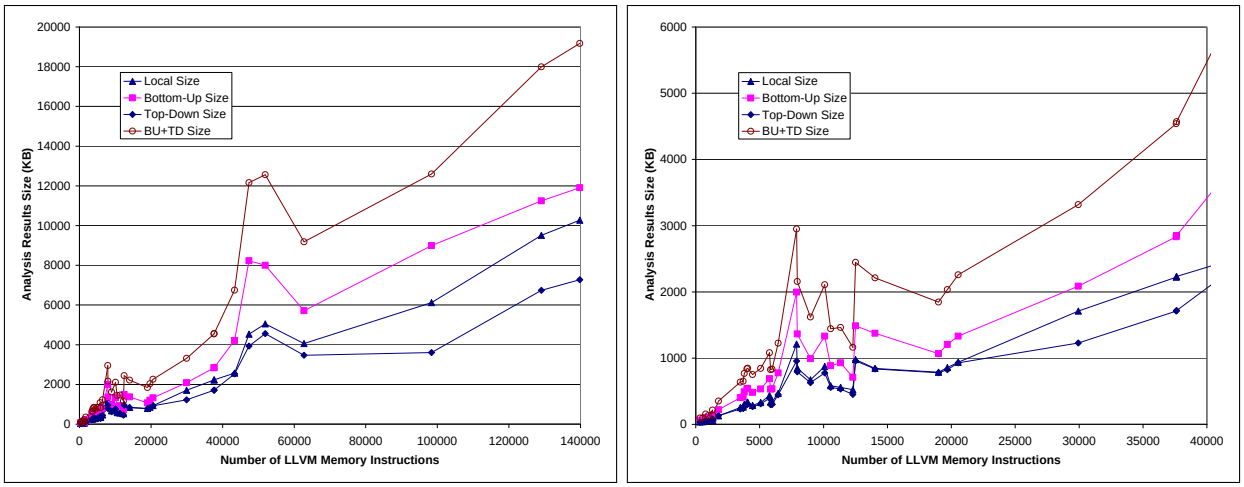


Figure 3.14: Scaling of Analysis Space with Program Size (Number of Memory Operations)
Left chart includes full data set. Right chart is zoomed in on lower-left quadrant.

In addition to being fast and compact, DSA is also very scalable. The Local and TD passes take roughly $O(n)$ time, where n is the number of memory operations in the program. Programs with a large number of globals (e.g., 254.gap, 253.perlbmk, 176.gcc, etc) show that the time required for the BU pass is related both to the program size and the number of globals in the program. We believe that a more aggressive form of the optimization described in Section 3.3.4 can be used to improve this, but even without further refinements DSA is extremely fast.

To put this into perspective, we compiled the 176.gcc, 253.perlbmk, and povray31 benchmarks with our system GCC compiler at the -O3 level of optimization. GCC takes 94.7s, 47.4s, and 38.5s to compile and link these programs, even though it does not contain *any link-time optimization* nor any compile-time interprocedural optimizations other than inlining. Given this, DSA takes only **3.4%**, **5.6%**, and **1.4%** of the total GCC compile times for these programs, despite the fact that GCC is not an aggressive interprocedurally optimizing compiler. We feel that this shows that DSA is fast enough for for use in realistic commercial compilers, particularly considering that it may be used for many varied clients (as described throughout this thesis).

3.4.3 Inferred Type Information

Figure 3.16 counts the number of load and store instructions ("accesses") whose pointer operand is determined to point to a *non-collapsed*, *complete* DS node: those that DSA is able to conclusively infer as type-safe. The first two columns list the name and total number of memory instructions

	Memory	Analysis Time (s)				Analysis Space (bytes)			
Benchmark	**Instrs**	**Local**	**BU**	**TD**	**L+B+T**	**Local**	**BU**	**TD**	**BU+TD**
176.gcc	139790	0.44	2.38	0.42	3.24	10275784	11906592	7270952	19177544
126.gcc	129083	0.44	1.87	0.37	2.68	9513864	11245056	6743568	17988624
253.perlbmk	98386	0.31	2.11	0.24	2.65	6125408	8996272	3601432	12597704
povray	62734	0.18	0.23	0.13	0.54	4062264	5716072	3470128	9186200
252.eon	51897	0.24	0.18	0.14	0.56	5048848	8005688	4556456	12562144
254.gap	47389	0.17	0.41	0.19	0.77	4534000	8233320	3928600	12161920
177.mesa	43352	0.10	0.06	0.07	0.23	2567048	4213688	2535080	6748768
147.vortex	37632	0.09	0.14	0.07	0.30	2232296	2852024	1718240	4570264
255.vortex	37601	0.07	0.10	0.09	0.26	2222704	2831328	1709200	4540528
134.perl	29940	0.08	0.12	0.06	0.26	1709904	2086760	1229320	3316080
099.go	20543	0.04	0.02	0.04	0.10	936608	1330576	927512	2258088
300.twolf	19686	0.05	0.01	0.03	0.09	857760	1207232	825944	2033176
namd	19002	0.05	0.02	0.02	0.09	786472	1067224	779768	1846992
186.crafty	14035	0.05	0.03	0.04	0.12	842968	1375824	836312	2212136
132.ijpeg	12507	0.03	0.02	0.02	0.07	978864	1487576	957488	2445064
bsim	12287	0.02	0.03	0.01	0.06	523968	709856	452680	1162536
146.wave5	11333	0.03	0.01	0.01	0.05	559304	929832	533936	1463768
188.ammp	10551	0.00	0.00	0.01	0.01	583504	885560	559200	1444760
197.parser	10086	0.04	0.03	0.03	0.10	874584	1332312	776144	2108456
175.vpr	8972	0.03	0.02	0.01	0.06	673488	992336	628608	1620944
124.m88ksim	7951	0.03	0.03	0.02	0.08	849928	1365728	792456	2158184
130.li	7894	0.03	0.09	0.04	0.16	1212472	1995008	955312	2950320
103.su2cor	6450	0.02	0.02	0.01	0.05	467496	777304	447472	1224776
173.applu	5966	0.02	0.01	0.01	0.04	343240	535856	298448	834304
110.applu	5854	0.01	0.02	0.00	0.03	336592	530208	295720	825928
104.hydro2d	5773	0.02	0.02	0.01	0.05	435680	688328	393360	1081688
168.wupwise	5087	0.01	0.01	0.01	0.03	328952	535112	309216	844328
145.fpppp	4447	0.01	0.01	0.01	0.03	285432	483256	267528	750784
172.mgrid	4064	0.01	0.01	0.01	0.03	335248	542752	304792	847544
107.mgrid	4010	0.02	0.01	0.01	0.04	331792	537104	301760	838864
101.tomcatv	3797	0.02	0.00	0.01	0.03	303168	489520	278928	768448
171.swim	3716	0.02	0.01	0.00	0.03	260736	411560	239616	651176
102.swim	3493	0.01	0.01	0.01	0.03	254136	404376	232176	636552
164.gzip	1785	0.00	0.01	0.00	0.01	128944	221712	129904	351616
183.equake	1340	0.00	0.00	0.00	0.00	57928	77744	57168	134912
256.bzip2	1315	0.01	0.00	0.00	0.01	83872	134040	81752	215792
181.mcf	991	0.00	0.00	0.01	0.01	55360	77880	53976	131856
179.art	773	0.00	0.00	0.00	0.00	61216	97536	57952	155488
fpgrowth	544	0.01	0.00	0.01	0.02	39240	59496	37656	97152
129.compress	326	0.00	0.00	0.00	0.00	34232	61784	32584	94368

Figure 3.15: DSA Analysis Time and Space Consumption Data

Benchmark	Mem Instrs	Safe Access	Unsafe Access	Safe Percent
SPEC CINT 2000				
181.mcf	991	556	7	98.8%
256.bzip2	1315	592	119	83.3%
164.gzip	1785	1086	38	96.6%
175.vpr	8972	3936	490	88.9%
197.parser	10086	1826	3199	36.3%
186.crafty	14035	8241	318	96.3%
300.twolf	19686	9882	1102	90.0%
255.vortex	37601	13403	9032	59.7%
254.gap	47389	11728	13196	47.1%
252.eon	51897	11038	14064	44.0%
253.perlbmk	98386	20145	37153	35.2%
176.gcc	139790	43504	30831	58.5%
average				69.6%
SPEC CFP 2000				
179.art	773	406	53	88.5%
183.equake	1340	620	86	87.8%
171.swim	3716	1829	372	83.1%
172.mgrid	4064	1937	470	80.5%
168.wupwise	5087	2829	338	89.3%
173.applu	5966	2968	676	81.4%
188.ammp	10551	2785	2961	48.5%
177.mesa	43352	4776	17429	21.5%
average				72.6%

Benchmark	Mem Instrs	Safe Access	Unsafe Access	Safe Percent
SPEC CINT 1995				
129.compress	326	187	30	86.2%
130.li	7894	1479	2155	40.7%
124.m88ksim	7951	3320	1536	68.4%
132.ijpeg	12507	4268	2517	62.9%
099.go	20543	11034	3	100.0%
134.perl	29940	5749	10677	35.0%
147.vortex	37632	13427	9032	59.8%
126.gcc	129083	38567	30103	56.2%
average				63.6%
SPEC CFP 1995				
102.swim	3493	1871	212	89.8%
101.tomcatv	3797	1982	281	87.6%
107.mgrid	4010	1914	462	80.6%
145.fpppp	4447	2673	481	84.7%
104.hydro2d	5773	2684	796	77.1%
110.applu	5854	2939	638	82.2%
103.su2cor	6450	3272	678	82.8%
146.wave5	11333	5203	2359	68.8%
average				81.7%
Other Programs				
fpgrowth	544	247	3	98.8%
boxed-sim	12287	2677	4226	38.8%
NAMD	19002	9229	686	93.1%
povray31	62734	13607	19722	40.8%
average				67.9%

Figure 3.16: Number of Load & Store instructions which access non-collapsed, complete, DS Nodes

(including address arithmetic, calls, etc) from Figure 3.12. The third column, labelled "Safe Access", is the number of load/store instructions that target non-collapsed complete nodes. The fourth column, labelled "Unsafe Access", is the number of load/store instructions which target either collapsed or incomplete nodes. The fifth column is the percentage of load/store instructions that are "safe".

The table shows that *many programs are found to be mostly type safe*, despite being written in languages that do not encourage disciplined use of types. For example, in CINT2000, 8 out of 12 of the programs are more than 50% type-safe, and 6/12 are more than 80% type-safe. The FP benchmarks generally do even better, due to simpler access patterns and data structures. For smaller and cleaner programs (e.g., those in the Olden suite [109]), many programs are fully 100% type-safe.

Of the programs that have a large number of non-type-safe accesses, the most common reason is the use of custom memory allocators (e.g., 197, 254, 176, and 253). If a program uses a custom memory allocator, DSA is not able to know that memory allocated through the custom allocator is disjoint from each other: this causes a large amount of node merging, and, if memory for different

types is involved, all of these nodes are collapsed.

This problem can be addressed by adding special attributes or pragmas to the programs (such as GCC's "attribute malloc") to indicate that the functions return disjoint memory, though our implementation of DSA currently does not support this. Note that, being a context-sensitive analysis, DSA is able to see through malloc "wrappers" like "xmalloc" and "operator new" without any special treatment of them; the problem occurs when the programmer *reimplements* the memory allocator. Note that this problem is not specific to DSA: all pointer analysis is affected, and the effect has been discussed in the literature before (e.g. [62]).

The second most common reason for a high percentage of non-type-safe nodes is due to the language runtime libraries for C++ and FORTRAN programs. Because these runtime libraries are compiled to LLVM and statically linked into the program[12] and they tend to use non-type-safe constructs (the FORTRAN runtime is written in C, for example). This is also important because the implementation details of some runtime functions causes data structures to be collapsed in the main program (e.g., `do_fio` in libf2c, which is used to perform non-type-safe file IO of scalars). A simple way to address this issue is to write transfer functions for each of the routines in these runtime libraries to more accurately describe the important points-to effects of each function.

Overall, we believe that these numbers show that a large amount of many programs written in C are type-safe, despite the fact that this property is not enforced by the language. Through the use of macroscopic techniques described in this thesis, we attempt to take advantage of this type information where it is available, without requiring the the entire program be written in a type-safe language, or be trivially type-safe (e.g., by disabling all unsafe operations).

3.5 Related Work

There is a vast literature on pointer analyses (e.g., see the survey by Hind [73]), but the majority of that work focuses on context-insensitive alias information and does not attempt to extract properties that are fundamental to macroscopic techniques (e.g., identifying disjoint data structure instances). For this reason, we focus on techniques whose goals are similar to ours.

[12] Note that the interprocedural optimizer is able to remove most of the obviously unused portions of these runtime libraries, through dead global and dead function elimination.

3.5.1 Shape Analyses

The most powerful class of related algorithms are those referred to as "shape analysis" [84, 60, 117]. These algorithms are strictly more powerful than ours, allowing additional queries such as "is a given data structure instance a singly-linked list?" However, this extra power comes at very significant cost in speed and scalability, particularly due to the need for flow-sensitivity and iteration [117]. Significant research is necessary before such algorithms are scalable enough to be used for moderate or large programs.

In contrast to shape analysis techniques, Data Structure Analyis is able to identify recursive data structures, but cannot determine whether something is a "doubly linked list" or "binary tree", and (because it uses flow-insensitive analysis) cannot make use of strong updates. Despite this, it is able to host a broad range of clients, such as those described throughout this work, and is efficient enough to be used on large programs.

3.5.2 Cloning-based Context-Sensitive Analyses

The prior work most closely related to our goals is the recent algorithm by Liang and Harrold [92], named MoPPA. The structure of MoPPA is similar to our algorithm, including Local, Bottom-Up, and Top-Down phases, and using a separate Globals Graph. For some programs, the analysis power and precision of MoPPA both seem very similar to Data Structure Analysis. Nevertheless, their algorithm has several limitations for practical programs. MoPPA can only retain field-sensitivity for completely type-safe programs, and otherwise must turn it off entirely. It requires a precomputed call-graph in order to analyze indirect calls through function pointers. It also requires a complete program, which can be a significant limitation in practice. Finally, MoPPA's handling of global variables is much more complex than Data Structure Analysis, which handles them as just another memory class. Both algorithms have similar compilation times, but MoPPA seems to require much higher memory than our algorithm for larger programs: MoPPA runs out of memory analyzing `povray3` with field-sensitivity on a machine with 640M of memory. In contrast, DSA can analyze the same program in less than one second and using less than 10MB of memory.

Ruf's synchronization removal algorithm for Java [114] also shares several important properties with ours and with MoPPA, including combining context-sensitivity with unification, a non-iterative

analysis with local, bottom-up and top-down phases, and node flags to mark global nodes. Unlike our algorithm, his work requires a call graph to be specified, it is limited to type-safe programs, and does not appear to handle incomplete programs.

Both the FICS algorithm of Liang and Harrold [91] and the Connection Analysis of Ghiya and Hendren [59] attempt to disambiguate pointers referring to disjoint data structures. But both ignore heap locations not relevant for alias analysis, and both algorithms have higher complexity.

Cheng and Hwu [25] describe a flow-insensitive, context-sensitive algorithm for alias analysis, which has three limitations relative to our goals: (a) they do not use cloning to represent distinct instances of memory objects allocated from the same program point (b) they represent only relevant alias pairs, not an explicit heap model; and (c) they use a k-limiting technique that would lose connectivity information for nodes beyond k links (instead of representing recursive structures with cycles). Additionally, they allow a pointer to have multiple targets (as in Andersen's algorithm), which is more precise but introduces several iterative phases and incurs significantly higher time complexity than our algorithm.

Deutsch [46] presents a powerful heap analysis algorithm that is both flow- and context-sensitive and uses access paths represented by regular expressions to represent recursive structures efficiently. Although based on access paths, it appears possible to reconstruct heap information from the regular expressions created. In practice however, his algorithm appears to have much a higher complexity than ours.

3.5.3 Non-cloning Context Sensitive Analyses

As discussed earlier, even many context-sensitive algorithms do not clone heap objects in different calling contexts. Instead, it is common to use more limited naming schemes for heap objects (often based on static allocation site[13]) [51, 143, 53, 138, 42]. This precludes obtaining information about disjoint data structure instances, which is fundamental to all applications of *macroscopic* data structure transformations. In the case of Figure 3.1, for example, all nodes of both lists are created at the same `malloc` site, which would force these algorithms to merge the memory nodes for the `X` and `Y` lists, preventing them from proving that the lists are disjoint.

[13]In principle, such algorithms can be implemented to use cloning, but the cost could become unbearably exponential [143, 53]. Making cloning efficient is the key challenge.

3.6 Data Structure Analysis: Summary of Contributions

Data Structure Analysis is a heap analysis algorithm designed to capture important properties of a program's memory usage (including connectivity, type-safety, mod/ref information, etc) to provide the foundation for all of the macroscopic analyses and transformations described in this thesis. The algorithm uses a combination of techniques that balance heap analysis precision (context sensitivity, cloning, and field sensitivity) with efficiency (flow-insensitivity, unification, and the globals graph) and includes important properties to make it usable by many clients (an explicit heap model, incompleteness information, mod/ref and composition information).

There are three key novel aspects to our algorithm, a key property that has been used but not articulated before, and a result which has not been achieved so far:

(i) We describe a collection of new algorithmic techniques which are needed to achieve scalable context-sensitive analysis. These techniques can potentially be applied to other context-sensitive algorithms (even non-unification based ones) to improve their analysis scalability. We show that DSA analyzes programs that are up to two hundred thousand lines of code in under 3.2 seconds, and uses very little memory.

(ii) The algorithm incrementally discovers an accurate call-graph for the program (and SCCs in the call graph) on-the-fly, using the call graph for parts of the analysis itself. The algorithm uses a novel extension of Tarjan's SCC finding algorithm permitting incremental discovery of SCCs in the call graph, even when edges are dynamically discovered and added.

(iii) The algorithm uses a simple mechanism (fine-grain incompleteness tracking) to solve several hard problems in pointer analysis, including the use of speculative type information, dynamic discovery of the call graph without iteration, and conservatively correct handling of incomplete programs. This allows it to analyze portions of programs safely and allows modular analysis of programs (e.g. analyzing portions of the program at compile-time and combining the graphs at link-time).

(iv) The property that we believe is fundamental to achieving a scalable "fully context-sensitive" algorithm is the use of a unification-based approach. With this combination, it is extremely

unlikely for the analysis representation to grow large, despite using a context-sensitive, field-sensitive representation. This is discussed in Section 3.2.5. Techniques that do not use unification (e.g., [140, 103]) have been shown to be scalable, but one or two orders of magnitude slower than those that do (e.g., DSA and [92]).

(v) Data Structure Analysis is efficient and scalable enough to achieve analysis times that are comperable to non-context-sensitive subset-based algorithms. This result indicates that, given a target analysis time budget, a compiler engineer can choose to implement either a context-sensitive unification-based algorithm (like DSA), or a non-context-sensitive subset-based approach (such as Andersen's algorithms with refinements). Finally, this scalability makes DSA (and other macroscopic techniques) efficient enough to be reasonable for inclusion in a commercial compiler. DSA is at least an order of magnitude faster than previous fully context sensitive algorithms, the first to be small fraction of the time required to compile the program with a standard optimizing compiler (in this case, GCC).

In addition to the research contributions, we describe the key engineering details that make the algorithm efficient and scalable in practice. These implementation details do not affect the theoretical time bounds of the algorithm, but can make the algorithm hundreds of times faster on some programs.

We showed that the algorithm is extremely fast in practice (taking less than 3.5s to analyze a program that is over 200K LOC), uses very little memory (less than 20MB on the same), and scales very well in analysis time and memory footprint for 40 benchmarks spanning 4 orders-of-magnitude of code size. Data Structure Analysis can be used to support a broad range of clients including the macroscopic applications described throughout this thesis as well as standard alias analysis and mod/ref clients, which is described and evaluated in Chapter 4.

Chapter 4

Using Data Structure Analysis for Alias and IP Mod/Ref Analysis

Data Structure Analysis is an aggressive memory analysis which is designed to be powerful enough to support the macroscopic techniques described in this thesis, but is also fully capable of supporting traditional alias and mod/ref based techniques. This chapter describes and evaluates **ds-aa**, an alias and mod/ref analysis implementation built using the DSA framework, with several example clients. The goal of this chapter is to show how a simple client is built using the DSA framework, described in Chapter 3, and show the alias and mod/ref precision provided by DSA compared against other analyses of similar compile-time cost. All of the evaluation in this section is performed in the context of the LLVM Compiler Infrastructure (Chapter 2).

4.1 Alias Analysis and Mod/Ref Information

The literature has thoroughly studied the computation and use of alias and mod/ref information. See, e.g., [73], for a survey of some of the available work in the field. In this section, we describe the context for this work and the assumptions we make. All of the alias analysis implementations described in this chapter are built in and follow the conventions of the LLVM Alias Analysis Framework [85].

Note that, in the LLVM compiler, all automatic (stack) scalar variables that do not have their address taken are promoted to SSA values, and are thus are not candidates for alias analysis (it is not possible to take the address of an SSA register). In LLVM, there are four operations that access memory: `load`, `store`, `call`, and `invoke`. See Section 2.2 for more details.

Alias analysis and mod/ref information are typically used by two very different forms of clients: optimizations and safety checking/program understanding tools. The two types of clients are characterized by how they use the resulting information and their tolerance for errors. An optimizing compiler requires the the pointer analysis be *safe* (i.e., it returns conservative information) while a checking and program understanding tools generally do not. Because the primary focus of this thesis is for program optimization, all analyses described and evaluated here (including DSA) are *conservatively correct*: If they cannot determine, for all executions of the program, that a statement is true, it does not assert it. For example, if it cannot prove that two pointers will never alias, it must return "MayAlias" (defined below).

4.1.1 Alias Analysis Assumptions and Applications

Alias analysis, in this context, is a static compiler analysis which performs some amount of up-front inspection of the program, builds data structures to summarize its results, then answers queries of the form "alias(P_1, S_1, P_2, S_2)", where P_1 and P_2 are pointers in the program and S_1 and S_2 are constant integers, which represent the size in bytes of the target of each pointer. This query can return one of three results:

- **MustAlias**: P_1 is always exactly equal to P_2.
- **NoAlias**: The two ranges $[P_1...P_1 + S_1)$ and $[P_2...P_2 + S_2)$ never overlap.
- **MayAlias**: The analysis can not prove that the result is either MustAlias or NoAlias (i.e., the ranges might overlap).

Alias analysis can support a wide variety of different clients, including devirtualization, common subexpression elimination, scalar promotion, etc. (even optimizations as simple as transforming `memmove` calls to `memcpy` calls if the source and destination ranges can never overlap). Figure 4.1 gives two examples to demonstrate how alias analysis can be used to prove the safety of redundant load elimination (a form of Common-Subexpression Elimination) and load hoisting (a form of Loop Invariant Code Motion). In Figure 4.1 (a) and (c), if an alias analysis can guarantee that P1 and P2 can never alias, CSE and LICM can transform the examples into the code in Figure 4.1 (b) and (d) respectively, which execute fewer dynamic loads from P1.

```
t1 = *P1;
*P2 = t2;
t3 = *P1;
```

(a) CSE Input

```
t1 = *P1;
*P2 = t2;
t3 = t1;      // load elim!
```

(b) CSE Desired Result

```
do {
    t1 = *P1;
    ... use t1
    *P2 = t2;
} while (...);
```

(c) LICM Input

```
t1 = *P1;      // hoisted!
do {
    ... use t1
    *P2 = t2;
} while (...);
```

(d) LICM Desired Result

Figure 4.1: Results of Example Pointer Analysis Clients

4.1.2 Mod/Ref Analysis Assumptions and Applications

Like alias analysis, mod/ref analysis is a well studied static compiler analysis which performs an up-front analysis, then responds to some number of client analyses. Our implementation supports two forms of mod/ref query. The first query is of the form "modref(I_1, I_2)", where I_1 and I_2 are two primitive operations in the program. This query can return one of several forms of dependence between the two operations, and supports general call/call mod/ref information, but is not described in detail for this work.

The second query is of the form "modref(I, P, S)", where I is a primitive operation, P is a pointer in the program, and S is a constant integer size. This query can return one of four possible results:

- **NoModRef**: I does not access the memory defined by the range $[P...P+S)$.
- **Ref**: I_1 might read the range $[P...P+S)$, but is guaranteed to not modify it.
- **Mod**: I_1 might modify the range $[P...P+S)$, but is guaranteed to not read it.
- **ModRef**: I_1 might modify or read the range $[P...P+S)$.

Mod/ref information can be used for a variety of purposes, such as dead store elimination, program slicing, and redundancy elimination. When used for redundancy elimination, mod/ref information is strictly more general than alias analysis information, as it allows the client to query about the mod/ref effect of function calls. Figure 4.2 gives two examples where mod/ref information for function calls allows the elimination of a potentially redundant load and the hoisting of a

```
t1 = *P1;
func();
t3 = *P1;
```

(a) CSE Input

```
t1 = *P1;
func();
t3 = t1;      // load elim!
```

(b) CSE Desired Result

```
do {
    t1 = *P1;
    ... use t1
    func();
} while (...);
```

(c) LICM Input

```
t1 = *P1;      // hoisted!
do {
    ... use t1
    func();
} while (...);
```

(d) LICM Desired Result

Figure 4.2: Example clients of mod/ref results

potentially loop invariant load from a loop. If the mod/ref analysis can prove that '`func`' never modifies P1 (i.e. the modref query returns NoModRef or Ref), it is legal for CSE to optimize (a) to (c) and LICM to optimize (b) to (d).

While computation and use of mod/ref information have been investigated in the literature, context-sensitive analyses tend to either be limited to cases with very simple aliasing [11, 36, 35] or too slow for practical use [83, 32, 130, 107, 97]. Because of this, use of context-sensitive mod/ref analyses (which permits aliasing) has largely been unattractive for inclusion in a commercial-grade compiler. Because DSA is very efficient and can directly provide context-sensitive mod/ref information, we feel is very important to consider it.

Note that mod/ref information nicely encompasses several ad-hoc optimizations performed by many compilers (e.g. optimizing "pure" and "const" functions, which do not access memory or only read memory), simplifies the implementation of many clients, and is more general than using traditional alias queries for many clients (such as redundancy elimination).

Note that it is possible to use a context-sensitive interprocedural data flow analysis post-pass to construct context-sensitive mod/ref information from a non-context-sensitive alias analysis [116], but we have not implemented and do not evaluate this option here.

4.2 Implementing Alias and Mod/Ref Analysis with DSA: `ds-aa`

The Data Structure Analysis algorithm described in Chapter 3 constructs several sets of graphs which capture a general-purpose abstraction of the program memory image. These graphs are

designed to represent important information about the memory usage of the program without tying the representation to a specific client. In this section, we describe `ds-aa`, an alias analysis implementation that uses the results of DSA to answer alias analysis queries.

4.2.1 Computing Alias Analysis Responses

DSA consists of three primary passes, each of which compute a set of graphs: the Local pass (Section 3.2.2), the Bottom-Up pass (Section 3.2.3), and the Top-Down pass (Section 3.2.4). Because DSA keeps track of what information is "complete" (see Section 3.1.1) at each stage of construction, we could use any of these three graphs to implement alias analysis.

In practice, we use the TD graphs for alias analysis, as they have the most complete information available in them: the graph for a function includes the effects of all callers and all callees, so the only incomplete information remaining is due to information that leaks in from outside of the analysis scope (e.g. memory which is passed to or returned from an external function). To answer an "alias(P_1, S_1, P_2, S_2)" query, `ds-aa` performs the following steps:

1. Look up the TD DSGraph G, which contains P_1 and P_2 in its E_V mapping.
2. Let the node/field pairs $\langle n_1, f_1 \rangle = E_V(P_1)$ and $\langle n_2, f_2 \rangle = E_V(P_2)$, using G's E_V mapping.
3. If $\mathbf{C} \notin flags(n_1)$ and $\mathbf{C} \notin flags(n_2)$, return MayAlias (if both nodes contain incomplete information, no judgement can be made).
4. If $n_1 \neq n_2$, return NoAlias (pointers point to two distinct nodes).
5. If not overlap(offsetof(f_1), offsetof(f_1)+S_1, offsetof(f_2), offsetof(f_2)+S_2) return NoAlias (if the fields cannot overlap, pointers point to distinct fields).
6. Return MayAlias.

Steps #1 and #2 perform simple map lookups to find the relevant information. Step #3 ensures that `ds-aa` is safe for incomplete programs: if both pointers point to non-complete nodes, no conclusion about them can be made. Note that if n_1 is complete and n_2 is not (or visa-versa), we know that the nodes are distinct and that n_1 can never be merged with n_2 no matter what code

is outside of analysis scope. If n_1 could ever be merged with n_2, it could not be marked complete, as described in Section 3.1.1).

Step #4 draws the conclusion that if the pointers point to distinct nodes (and if at least one is marked complete, due to step #3), the pointers can never alias. Step #5 uses field sensitivity to refine the alias analysis in the case when the pointers point to the same node. In this case, if the two fields do not overlap, `ds-aa` can conclude NoAlias. If neither Step #4 or #5 are able to determine non-aliasing, `ds-aa` must return MayAlias.

Notice that DSA is incapable of returning must alias information. In particular, even if $n_1 = n_2$ and $f_1 = f_2$, DSA cannot prove that both pointers point to the same dynamic memory object, only that they are in the same class (for example, it cannot determine that the pointers point to the exact same linked list node). If a node only contains **G**lobal information (no heap, stack or unknown memory), we could conceptually provide must alias information in cases where our aggregate model does not make this unsound (e.g. we collapse an entire array to one element). We have not investigated this possibility.

4.2.2 Computing Mod/Ref Responses

The steps required to compute a safe answer to the "modref" queries described in Section 4.1.2 depend on the the different instructions passed in as arguments. As mentioned above, there are 4 operations that (directly or indirectly) can access memory: load, store, call, & invoke. The LLVM framework handles the simple mod/ref queries automatically (e.g., an add operation mod/refs nothing), and dispatches the remaining queries to the "alias" query above (e.g. to determine if a store mods a location being loaded), to a call/call dependence tester, or to the second second modref query above which checks a call against a memory range (e.g. to test a load against a call).

DSA has all of the information it needs to compute context-sensitive mod/ref information for function calls. In particular, the Bottom-Up graphs capture the direct and indirect mod/ref effects of calling the function, in any context, at a per DSNode granularity. While this information is general enough to even allow testing for call/call dependence, none of our clients currently use this information. As such, we only describe call/location mod/ref analysis here.

To respond to a "modref(I, P, S)" query, where I is a call or invoke, `ds-aa` performs the

following steps:

1. Look up the Top-Down DSGraph G, that includes the function containing I and P in its E_V mapping.

2. Let the node/field pair $\langle n, f \rangle = E_V(P)$, using G's E_V mapping.

3. If $\mathbf{C} \notin flags(n)$, return ModRef (*memory is incomplete, cannot draw a conclusion if F accesses it*).

4. Let AC be the set of actual callees for I. If the actual callees are unknown, return ModRef.

5. Remove any external functions from AC.

6. If AC is empty, return NoModRef (AC *must have been empty*[1] *or contained only external functions. Since the memory does not escape the program, external functions cannot mod/ref it*).

7. *Union together all of the Bottom-Up graphs for the callees, merging the corresponding formals for each function:*
   CG = Empty DS Graph
   $\forall F \in AC$
      cloneGraphInto(BUDSG(F), CG)
      mergeArguments(F, CG)

8. Compute the mapping M from nodes in G to nodes in CG, as defined by the actual argument/formal argument bindings defined by I, and mutual global variables defined in G and CG.

9. *Check nodes from the BU Graphs for mod/ref flags:*
   ModRefResult $R = \{\}$
   $\forall n_{CG} \in M(n)$
      if ($\mathbf{M} \in flags(n_{CG})$) $R = R \cup \{Mod\}$

[1] If the set of actual callees for a function is empty, the call site must be dynamically unreachable.

if ($\mathbf{R} \in flags(n_{CG})$) $R = R \cup \{Ref\}$

Return R

Steps #1 and #2 perform simple lookups to get the information we need. Step #3 checks for incomplete information: if P points to incomplete memory, we do not draw any conclusion about it. Step #4 computes the actual callees for a calle site (note that our implementation currently only implements direct calls, but we could easily add support for indirect calls). Step #6 implements a trivial form of mod/ref analysis that any conservatively-correct whole-program analysis can provide: calls to external functions are known to not access memory that does not escape from the program[2].

Given a direct call to a function in the program, Step #8 computes the relevant mapping from nodes in TD Graph G to the nodes in CG graph. Because the bottom-up graphs for the functions in AC were inlined into the caller graph, we know that the caller graph is at least as constrainted as the callee graphs (and may be more so). As such, we compute (and cache) the mapping from nodes in G to nodes in CG defined by the call site I. Finally, Step #9 iterates over all of the nodes that n maps to in the CG graph, and unions together the mod/ref information from these nodes to form a result. Note that if n is never accessed by F, it will not map to any nodes, thus we will compute a NoModRef result.

Note that DSNodes in CG only track mod/ref information on a per-node basis. DSA could be trivially extended to support more precise mod/ref information by tracking mod/ref information on a per-field basis. To do this, we expand the **M** and **R** bits to be bit-vectors that tracks one bit for every field in a node. This would have slightly higher overhead than tracking one bit per node, but would improve mod/ref precision for programs that use structures heavily.

`ds-aa` Mod/Ref precision could also be improved for nodes that escape the program. In particular, even if a node is not marked complete (which is handled above by Step #3), a call does not mod/ref the node if all of the DS Nodes mod/ref'd by the call are complete. The check for Step #3 above could be enhanced to take this into consideration.

[2]Note that this judgement relies on the assumption that the externally called function cannot make a direct call back into the program. This assumption is guaranteed by standard "whole program" optimziation flags offered by many aggressive compilers, and is always safe for the programs in our testsuite.

4.3 Alias Analysis Implementations for Comparison

In order to evaluate the effectiveness of DSA for standard alias analysis clients, we need to compare it against the precision of other well-known algorithms. In this section, we briefly describe the intraprocedural (**local**), and the three interprocedural algorithms (**steens-fi**, **steens-fs**, and **anders**) that we compare **ds-aa** to. Note that the relative precision of Steensgaard's and Andersen's algorithms have been characterized by other studies in the past (e.g., [72]): we chose this combination of analyses as a way to evaluate how the various design decisions impact the precision of DSA (including context-sensitivity, field-sensitivity, and the choice of a unification based approach).

Note that all of these analyses handle incomplete programs in a conservatively correct manner. Also, we are careful to use the same set of function stubs for known external functions with each of the interprocedural algorithms.

4.3.1 `local` Alias Analysis

The **local** alias analysis is an aggressive local analysis which attempts to disambiguate pointers with a large collection of ad-hoc rules (this is the LLVM "-basicaa" pass). For example, it knows "A[i]" doesn't alias "B[i]" if "A" and "B" are two different global, stack, or heap objects. It knows that "A[1]" doesn't alias "A[2]", "A->field1" doesn't alias "A->field2", knows alias and mod/ref properties of automatic variables without their "addresses taken"[3], etc.

The **local** analysis also provides mod/ref information for standard C library functions never read or write memory (such as "`sin`" and "`cos`"), and those that only read memory (such as "`strcmp`" and "`strlen`"). As of this writing, it does not model functions that may modify `errno` or other memory (such as "`sqrt`" and "`log`"). It is also smart enough to know that "const" globals can never be modified.

While the **local** analysis is extremely fast and very simple, it is able to provide a large amount of alias information, particularly for codes that make heavy use direct accesses to global and local variables. As others have observed [62], it makes the most sense to use an aggressive local analysis *in combination* with interprocedural techniques in most settings. For this reason, the LLVM alias

[3]Note that, in practice, this only occurs for aggregates like structs or arrays. Scalar variables are promoted to SSA values as described in Section 4.1.

analysis framework supports chaining of analyses together: if one analysis cannot answer a query precisely, the next analysis in the chain is queried and so on.

In this evaluation, when any of the four interprocedural analyses (**steens-fi**, **steens-fs**, **anders**, **ds-aa**) are unable to resolve a query, they chain to the **local** algorithm. Thus, when comparing the interprocedural algorithms, the **local** algorithm is the baseline. This avoids overstating the contribution of the interprocedural analyses.

4.3.2 `steens-fi` Alias Analysis

Steensgaard's flow-insensitive, context-insensitive, and field-insensitive alias analysis [129] is a well known algorithm that computes an approximation of the heap in linear space and almost linear time. It uses Tarjan's union-find data structure to efficiently partition memory objects into equivalence classes. This algorithm is extremely fast, but produces a coarse approximation of the heap.

We name our implementation of Steensgaard's algorithm **steens-fi**, and implement it using the DSA framework. In particular, we use the standard DSA local analysis phase, then merge all of the computed graphs into a single graph for the whole program, then perform actual/formal argument binding. Because we want to evaluate a field-insensitive version of Steensgaard's algorithm, we artificially collapse all nodes in the resultant graph to discard any field sensitivity captured by DSA.

As a result, our **steens-fi** implementation differs from Steensgaard's algorithm in two ways: First its uses the standard DSA completeness tracking to make the analysis result sound for incomplete programs. Second, it keeps the mod/ref bits for memory objects, allowing it to make judgements about memory that is either never read or never stored to. This information can occasionally be used to mark global variables 'const' if they are never modified, for example.

4.3.3 `steens-fs` Alias Analysis

The **steens-fs** alias analysis is identical to **steens-fi**, except that it does not artificially collapse nodes in the resultant graphs. This produces a field-sensitive variant of Steensgaard's analysis, similar in spirit to that described in [128].

4.3.4 `anders` Alias Analysis

The **anders** alias analysis is a simple implementation of Andersen's flow-insensitive, context-insensitive, field-insensitive subset-based pointer analysis [6]. It is strictly more powerful than **steens-fi**, and while the worst-case complexity is $O(n^3)$, with refinements [52, 113, 105], it can be made to run very fast in practice.

Our implementation of Andersen's analysis is accurate, but lacks the key refinements which make it efficient in practice. Because our implementation is very slow, we do not compare the analysis time of our implementation against DSA or any other algorithm. We believe that a well-engineered implementation of a context-insensitive Andersen's analysis should require analysis time comparable to the analysis time used by DSA (e.g. seconds for programs that are hundreds of thousands of lines of code).

The only difference between our implementation of Andersen's analysis and the standard formulation is the introduction of a "universal" node, which represents information flow into and out of the program. The universal node is a distinguished memory location which points to itself. After constraint solving, any pointers that target the universal node are known to point to memory that escapes the program, allowing conservative whole-program analysis. Our implementation also explicitly tracks pointers to the null object (the virtual object whose address is the null pointer). This allows us to track which pointers may point to null.

4.4 Analysis Precision with a Synthetic Client

In order to evaluate the precision of an alias analysis, we simply execute some number of clients on the full suite of benchmarks introduced in Section 3.4.1 with each of the analyses we are evaluating. Clearly it is infeasible to evaluate all possible alias scenarios in this study, so we focus on two here. This section evaluates the precision of `ds-aa` and the other alias analysis implementations with a synthetic client, which attempts to compute raw alias and mod/ref analysis precision metrics. Section 4.5 evaluates the analyses using a specific client, which performs a suite of loop memory optimizations such as hoisting loads, promoting memory to a register, etc.

This synthetic client evaluated in this section attempts to examine the precision of all alias

information that could possibly be used by an intraprocedural client. Because we cannot evaluate precision of these algorithms for all possible clients, this experiment aims to provide a reasonable metric which can be used to evaluate suitability for standard intraprocedural clients which make queries such as those described in Section 4.1. All of the evaluation in this section is performed in the context of the LLVM Compiler Infrastructure (Chapter 2).

Evaluating the precision of alias analyses is very difficult if the different analyses have widely varying implementation details. For example, many papers use the size of "points-to sets" to evaluate the precision of an analysis: the smaller the set the better. This metric works reasonably well if the analysis implementations all use the same system for naming memory objects in the program, but produces incomperable results otherwise. In particular, context-sensitive analyses may clone an object multiple times: two pointers may point to memory that is allocated at the same source line, yet the analysis can determine the pointers never alias.

Because of this difficulty, and because we don't want the client to know anything about the implementation of the pointer analysis, we use a different approach. LLVM includes a synthetic alias analysis client "AA-EVAL", which evaluates alias and mod/ref precision of an arbitrary pointer analysis implementation. It contains two phases: the first gathers alias analysis information the second gathers mod/ref information.

4.4.1 Alias Precision

In order to evaluate alias analysis precision, the AA-EVAL client iterates over each function in the program. Within each function, AA-EVAL builds a set of pointers that are used by the various memory accesses in the body of the function (e.g. by load and store instructions). Given this set of instructions, it does a simple $O(N^2)$ alias query of every pointer against all of the others[4] and counts the alias responses. Because the MayAlias response is the only response that indicates lack of information, an analysis with a lower may alias response percentage is more precise than one with a higher percentage of may alias responses.

This portion of the AA-EVAL client produces a metric that is very similar to the "alias frequencies" described in [42]. The primary difference between that work and this evaluation is that they

[4]Because alias relations are symmetric [(alias(X, Y) = alias(Y, X)] and a pointer always must-aliases itself [alias(Z, Z) = MustAlias], AA-EVAL only performs $N^2/2$ queries.

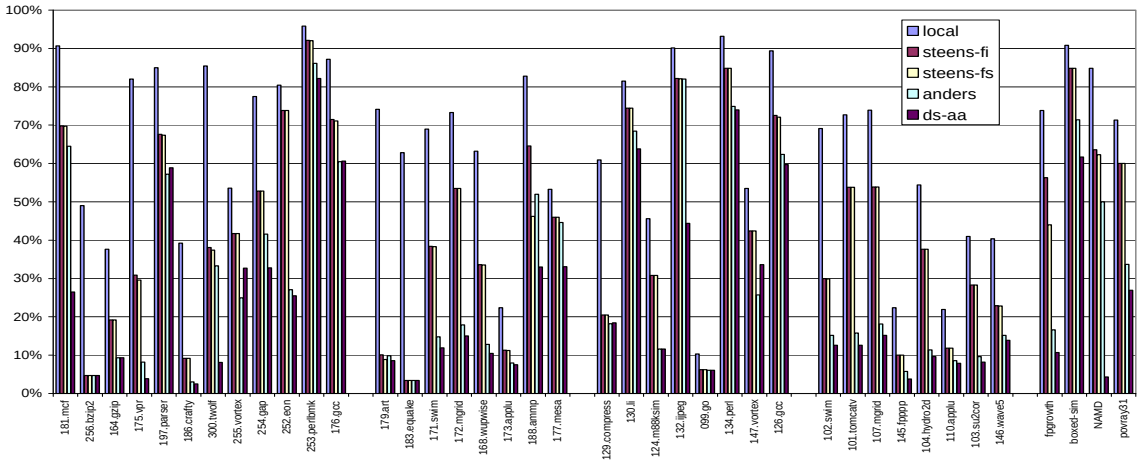


Figure 4.3: Percent of AA-EVAL Alias Queries Returned "May Alias"

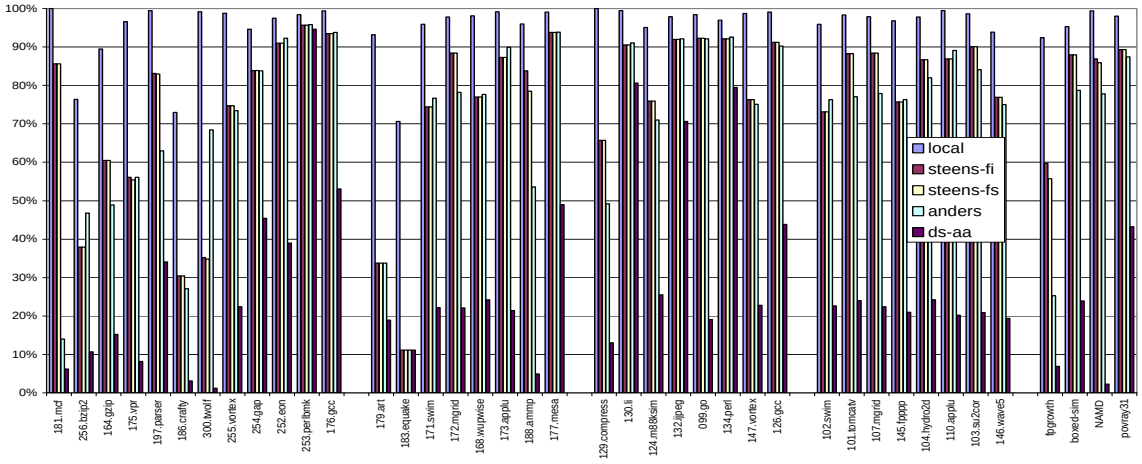


Figure 4.4: AA-EVAL Mod/Ref Query Responses of "May Mod or Ref"

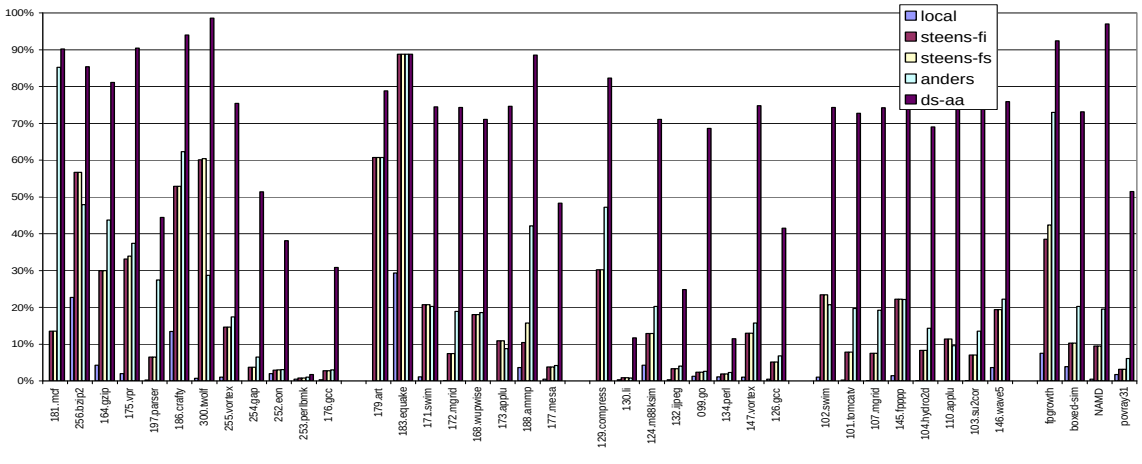


Figure 4.5: AA-EVAL Mod/Ref Query Responses of "No Mod or Ref"

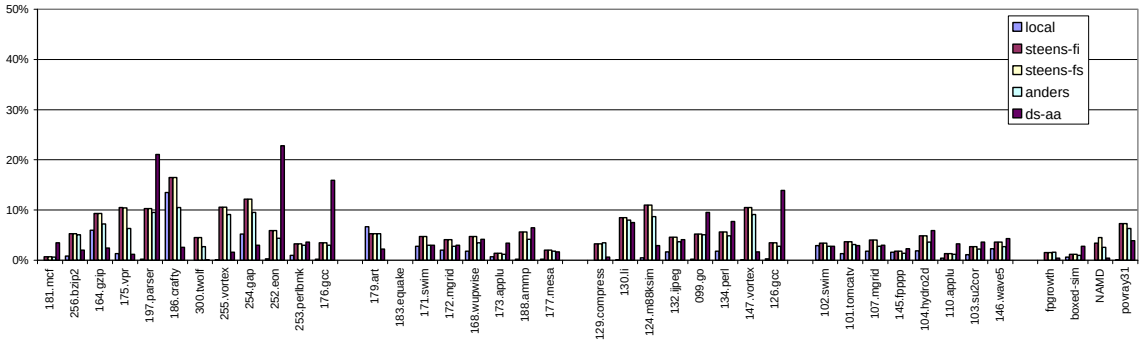


Figure 4.6: AA-EVAL Mod/Ref Query Responses of "May Only Ref"

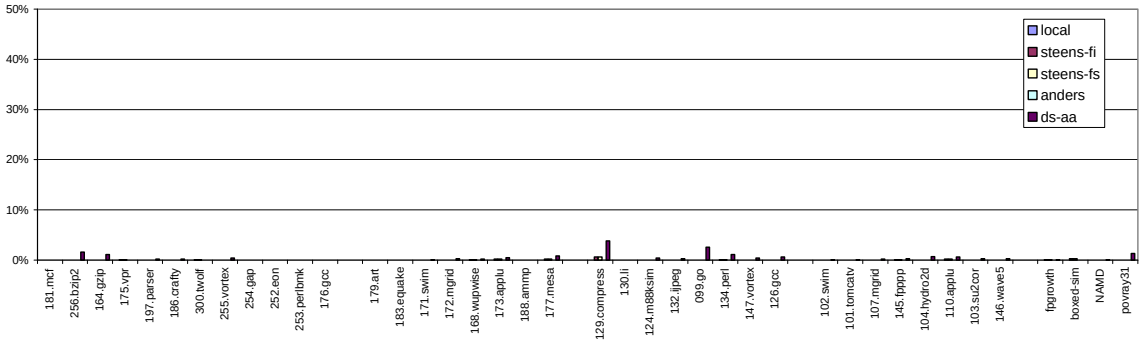


Figure 4.7: AA-EVAL Mod/Ref Query Responses of "May Mod Only"

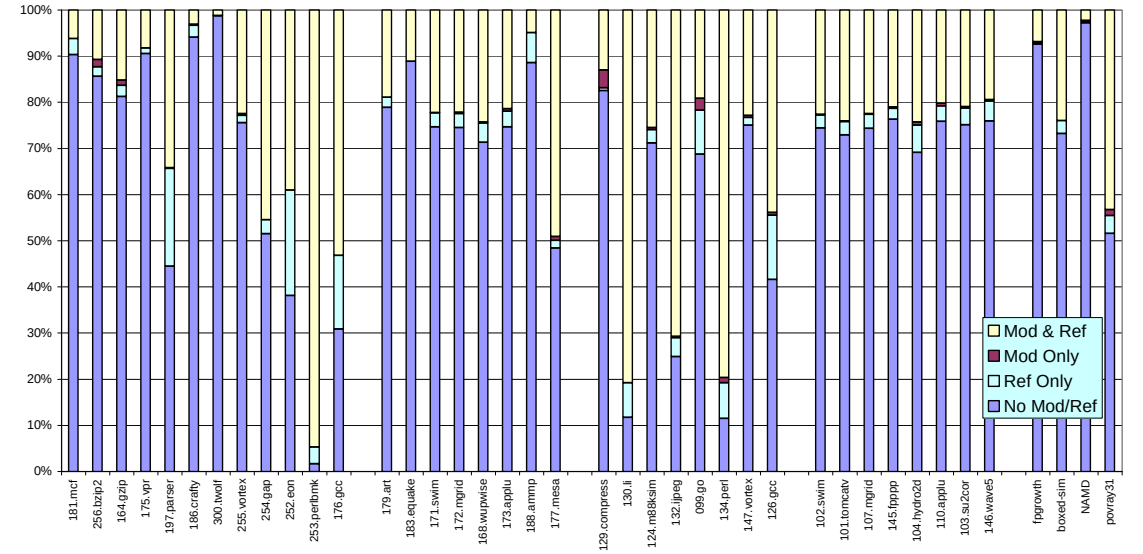


Figure 4.8: AA-EVAL Mod/Ref Query Responses for ds-aa

only consider one level of pointer dereference, where we consider all levels. For example, for the statement "*p = **q", we would count all alias pairs <*p,*q>, <*q,**q>, and <*p,**q>, where Das et.al., only count the last. A secondary difference is that we consider must-alias information to be accurate, they only count no-alias as a precise response. We believe this second difference to be very minor as the only analyses capable of returning must alias information in this evaluation are the **local** and **anders** analyses, which should not impact the evaluation of the DSA-based analyses.

Figure 4.3 shows the percentage of AA-EVAL queries that return a MayAlias response for each of the benchmarks in our suite and for each alias analysis implementation. All of the charts in this section are grouped by benchmark suite and ordered according to the number of memory instructions in the program (to match tables in Section 3.4). Thorough inspection of this figure confirms and validates several properties of pointer analyses which have been previously discussed in the literature, and shows that DSA provides very accurate points-to information in addition to being able to support the macroscopic techniques described in this thesis.

- *Trivial local analysis can successfully resolve a large number of queries*, particularly in simple array-based programs that do not pass values heavily by reference [62]. In particular, three FORTRAN programs have over 75% of their alias queries disambiguated without *any* interprocedural analysis at all, and 10 programs across the suite have over 50% of their alias queries resolved by the **local** algorithm. We believe that this shows the importance of evaluating interprocedural analyses together with a local algorithm, to avoid overstating the contribution of the interprocedural technique.

- *Any interprocedural analysis is far better than none* in many cases (e.g., 256.bzip2, 186.crafty, 175.vpr, 179.art, and 129.compress), even if it is as simple as Steensgaard's imprecise (but very fast) analysis. This argues for *every compiler* implementing some form of interprocedural pointer analysis if possible. Because Steensgaard's algorithm is the most straight-forward to implement, and has an excellent worst-case complexity in its simplest form, it should probably be the best candidate for an implementor who does not want to invest much time in pointer analysis.

- *Field sensitivity can substantially improve the precision of unification-based analysis in pro-*

grams that use multiple instances of structures with different types. While it makes no precision difference for a large number of programs, **steens-fs** is reasonably more precise than **steens-fi** for 188.ammp, fpgrowth, 175.vpr, 300.twolf, 176.gcc, 179.art, NAMD, povray, and for a large number of smaller programs that are not included in this data set (e.g. the Olden suite). If implementing a unification-based approach, adding field sensitivity should be considered. Note that **steens-fs** is more precise than **anders** for 188.ammp, due to the large contribution of field sensitivity.

- All other factors being equal, *subset-based analysis is far more precise than unification-based analysis.* While it is clear from the formulation that subset-based analysis is *at least* as precise as unification-based analysis, the numbers show that in many cases, a subset-based analysis (such as **anders**) if far superior *in practice.* Given a choice between implementing basic Steensgaard's algorithm and Andersen's algorithm, and given the resources to implement all of the refinements to make Andersen's algorithm scalable in practice, Andersen's should be far preferred.

- *Adding context sensitivity to a unification-based pointer analysis can allow it to meet or exceed the precision of a subset-based analysis in most cases.* Others have shown that either limited (e.g., [41]) or full (e.g., [92]) context sensitivity can be used to achieve this added precision. Our experience (matching other researchers [41, 53]) is that bidirectional argument binding is the leading cause of precision loss in a unification-based analysis. This problem can either be solved *either* by using context sensitivity, or a subset-based analysis. Note that adding context sensitivity to a subset-based analysis has been shown to only provide a marginal increase in precision [55] and can be impractically expensive [103, 102].

- *Using a cloning-based context-sensitive analysis can yield far more accurate points-to results than using a static naming scheme for heap and stack objects* [92, 103, 140]. The effect is most pronounced in programs that use a large amount of heap allocated data and have few static allocation sites. For example, the 175.vpr, 300.twolf, and 252.eon programs which have simple wrapper functions around malloc that prevent the context-insensitive algorithms from detecting the independence of any memory allocated from these wrappers. While special

purpose tricks [62] can be used to address this problem in limited cases, only full context sensitivity can address the problem in its full generality. Note that context sensitive algorithms that name heap objects by their static allocation site will suffer the same precision problems as context-insensitive algorithms for such programs.

Overall, these numbers show that the raw alias disambiguation precision of DSA is comparable to Andersen's algorithm in many cases (256.bzip2, 164.gzip, 183.equake, 176.gcc, 129.compress, etc), only occasionally slightly worse (197.parser, 255.vortex), and far better in several (181.mcf, 175.vpr, 186.crafty, 300.twolf, 172.mgrid, fpgrowth, NAMD, etc). Cases where Andersen's algorithm is more precise than DSA show cases where the precision advantage of a subset-based (instead of unification-base) approach out-weigh the precision advantage of using a context-sensitive (instead of a context-insensitive) approach.

4.4.2 Mod/Ref Precision

To evaluate the precision of mod/ref information returned by an implementation, the AA-EVAL client iterates over each function in the program, builds the list of pointers used in the function, and collects a list of all of the function calls in the body of the function. It then performs $O(M * N)$ mod/ref queries (to determine whether the analysis can decide whether a function call can modify or read the memory location) and counts the frequencies of the various results. Note that AA-EVAL only queries mod/ref information of locations against calls, it does not query for call/call dependence information.

Figures 4.4, 4.5, 4.6, and 4.7 evaluate the various AA-EVAL response percentages for each benchmark in our suite, and Figure 4.8 shows the composite results for just DSA in one figure (as a different way of visualizing the DSA data).

The results demonstrate several aspects of our analysis implementations and how they provide mod/ref information:

- The **local** analysis is capable of providing mod/ref information for a wide range of standard C library functions (e.g. `sin` and `cos`). Programs that have a high percentage of calls to standard library functions (e.g. 183.equake) are well served by this information.

- Any interpocedural pointer analysis can give good mod/ref information when a query asks about an external function call and memory that is known not to escape the program (as implemented by step #6 of the `ds-aa` mod/ref implementation, and replicated in the other interprocedural analyses). For example, Figure 4.5 shows that even **steens-fi** is able to resolve most of the mod/ref queries for programs that use a large number of external function calls (such as 179.art, 183.equake) which are not modelled by the **local** analysis. This can occur either because they are not part of libc, or because they may modify memory (e.g. `sqrt`, which can modify `errno`).

- Failing the two cases above, **steens-fi** and **steens-fs** can occasionally provide mod/ref information for memory locations that are never read or never written in the program. For example, this can occur when a global variable is logically const, but not marked as such (the **local** analysis takes care of the case when it is marked const). If memory is never stored to, it is trivial to see that no stores or calls can modify it. Note that it should be possible to extend the **anders** analysis to incorporate this information if desired: we included it in the **steens** implementations because the local pass of DSA provides the information for free.

- DSA's direct support for context-sensitive mod/ref information makes it far more precise than any of the other algorithms for all forms of mod/ref information, which can be seen in Figures 4.4 and 4.5. In 24 of the 40 programs, **ds-aa** is able to return NoModRef 40% more often than **anders**, and often does significantly better than that (e.g. resolving 75% of the queries in NAMD as NoModRef and 50% more queries for all the programs in SPEC FP95). This is particularly significant because **ds-aa** requires analysis time comparable to a well-tuned **anders** implementation.

The results show that DSA (like other context-sensitive algorithms) clearly yields more accurate mod/ref information than non-context-sensitive algorithms (i.e. there is a reduction of "mod and ref" results and an increase in "Not mod or ref" results).

Despite this, the mod/ref precision of DSA can still be improved in two ways: first, we could track mod/ref information by-field instead of by-node. Second, our `ds-aa` implementation could be extended to support mod/ref queries for indirect function calls, which would improve precision

for programs like 253.perlbmk which include a large number of indirect calls.

4.5 Analysis Precision with Scalar Loop Optimizations

The second client we evaluate, the LLVM LICM pass, performs a small collection of scalar loop optimizations. It optimizes scalar operations using standard SSA and loop analyses, extending them to load and store operations when mod/ref analyis can prove that it is safe. In particular it performs: a) hoisting of load instructions to the loop preheader, b) sinking of load instructions to loop exit blocks, c) promotion of stores in a loop to use a temporary and load/store once outside the loop. Figure 4.9 gives examples of the transformations applied.

```
do {
    t1 = *P1;
    ... use t1
    *P2 = t2;
} while (...);
```

(a) Load Hoisting Input

```
t1 = *P1;       // hoisted!
do {
    ... use t1
    *P2 = t2;
} while (...);
```

(b) Load Hoisting Result

```
do {
    t1 = *P1;
    ...
    *P2 = t2;
} while (...);
... use t1
```

(c) Load Sinking Input

```
do {
    ...
    *P2 = t2;
} while (...);
t1 = *P1;       // sunk!
... use t1
```

(d) Load Sinking Result

```
do {
    t1 = *P1;
    ... use t1
    *P2 = t2;
    ...
    *P1 = t3;
} while (...);
```

(e) Register Promotion Input

```
tmp = *P1;      // Promoted!
do {
    t1 = tmp;   // Promoted!
    ... use t1
    *P2 = t2;
    ...
    tmp = t3;   // Promoted!
} while (...);
*P1 = tmp;      // promoted!
```

(f) Register Promotion Result

Figure 4.9: Scalar Loop Optimization Transformations

The figure shows that we are investigating the effects of two forms of load motion (hoisting and sinking) and register promotion [37] (also known as "Location Invariant Code Motion [61]). Figure 4.9(a) shows a simple example of a loop invariant load. This load may be safely hoisted

out of the loop (producing the code in (b)) if mod/ref analysis is able to prove that nothing in the body of the loop can modify the value of *P1 (for example, P1 and P2 don't alias). Hoisting the load out of the loop reduces the dynamic number of loads if the loop executes more than once.

Figure 4.9(c) shows the same example, but the loaded value is only used outside of the loop. In this case, if the the loaded value is not modified between the load and all exits of the loops, it can be sunk to the loop exits, producing the code in (d). This reduces the number of dynamic executions of the load if the loop iterates more than one time.

Finally, Figure 4.9(d) illustrates a loop that has loads and stores to a loop invariant address. In this case, if the only accesses (mods and refs) to the memory are through must-aliased pointers, the memory location can be promoted to a temporary which is eligible for register allocation. This transformation reduces the number of dynamic loads and stores inside of the loop, which reduces memory traffic if the loop iterates more than one time.

4.5.1 Number of Transformations Performed

We compare the relative effectiveness of our various analyses by running this set of optimizations with each analysis, and comparing the number of transformation that are performed. In particular, we count three numbers here: 1) the number of memory locations promoted to a register, 2) the number of load instructions hoisted or sunk, and 3) the number of non-load instructions hoisted or sunk out of the loop. #1 and #2 are described above. #3 is the number of non-memory operations that are removed from the loop, which is limited by the number of memory operations that are hoisted (e.g. if *P is hoisted from the loop, the division operation in (*P)/100.00 can be hoisted). Figure 4.10 counts the number of register promotion transformations performed with each analysis, Figure 4.11 lists the number of loads hoisted or sunk, and Figure 4.12 counts the number of instructions hoisted or sunk with each analysis.

We perform this evaluation on the programs in each suite after the programs have undergone standard compile and link-time optimization, including whole program inlining, interprocedural constant propagation, etc. The link-time optimizer does run all of these optimizations, including LICM, with the `local` alias analysis, so most of the opportunities for motion and promotion that are achievable with the local analysis have already been performed (those few that are missed are

Benchmark	Transformation Count					Transformation Ratio	
	local	steens-fi	steens-fs	anders	ds-aa	ds-aa/st-fs	ds-aa/and
SPEC CINT 2000							
181.mcf				2	2	inf	1.00
256.bzip2		23	23	23	23	1.00	1.00
164.gzip		13	13	13	16	1.15	1.15
175.vpr		17	17	17	19	1.12	1.12
197.parser		3	3	3	3	1.00	1.00
186.crafty		14	14	14	17	1.21	1.21
300.twolf	1	62	62	86	139	2.24	1.62
255.vortex		2	2	28	46	23.00	1.64
254.gap		26	26	30	55	2.12	1.83
252.eon		7	7	12	77	11.00	6.42
253.perlbmk		10	10	21	18	1.80	0.86
176.gcc		27	27	28	53	1.96	1.39
SPEC CFP 2000							
179.art		3	3	3	3	1.00	1.00
183.equake							
171.swim					1	inf	inf
172.mgrid							
168.wupwise	4	4	4	4	24	6.00	6.00
173.applu		2	2	17	7	3.50	0.41
188.ammp		39	39	71	81	2.08	1.14
177.mesa		3	3	3	3	1.00	1.00
SPEC CINT 1995							
129.compress					1	inf	inf
130.li		6	6	5	51	8.50	10.20
124.m88ksim		14	14	20	64	4.57	3.20
132.ijpeg		2	3	3	6	2.00	2.00
099.go		4	4	5	13	3.25	2.60
134.perl		10	10	9	17	1.70	1.89
147.vortex		2	2	28	46	23.00	1.64
126.gcc		23	23	26	57	2.48	2.19
SPEC CFP 1995							
102.swim					1	inf	inf
101.tomcatv					3	inf	inf
107.mgrid							
145.fpppp		22	22	24	1573	71.50	65.54
104.hydro2d		19	19	20	23	1.21	1.15
110.applu		2	2	17	7	3.50	0.41
103.su2cor		11	11	17	54	4.91	3.18
146.wave5		4	4	4	16	4.00	4.00
Other Programs							
fpgrowth		2	2	3	3	1.50	1.00
boxed-sim		30	30	25	45	1.50	1.80
NAMD		15	15	27	47	3.13	1.74
povray31	1	17	26	140	228	8.77	1.63

Figure 4.10: Number of Memory Locations Promoted To Registers

We elide zeros from the table.

Benchmark	local	steens-fi	steens-fs	anders	ds-aa
SPEC CINT 2000					
181.mcf		14	14	14	14
256.bzip2	1	16	16	16	20
164.gzip	1	8	8	8	12
175.vpr		386	386	462	542
197.parser		33	33	42	47
186.crafty		45	45	46	124
300.twolf	1	373	373	441	592
255.vortex	1	86	86	173	300
254.gap		22	22	30	142
252.eon		51	51	73	307
253.perlbmk		17	17	32	21
176.gcc	3	319	319	332	467
SPEC CFP 2000					
179.art		73	73	73	73
183.equake	1	60	60	60	60
171.swim		3	3	3	11
172.mgrid		5	5	5	16
168.wupwise	6	13	13	16	41
173.applu		3	3	8	10
188.ammp		17	17	56	69
177.mesa		54	54	57	78
SPEC CINT 1995					
129.compress					1
130.li					15
124.m88ksim		6	6	7	20
132.ijpeg		60	62	91	148
099.go	1	6	6	6	30
134.perl		31	31	46	37
147.vortex	1	86	86	173	300
126.gcc		276	276	332	455
SPEC CFP 1995					
102.swim		3	3	3	11
101.tomcatv		5	5	7	9
107.mgrid		5	5		13
145.fpppp		16	16	18	42
104.hydro2d		9	9	31	38
110.applu		3	3	8	10
103.su2cor		10	10	19	78
146.wave5		3	3	35	51
Other Programs					
fpgrowth		9	10	21	27
boxed-sim		28	28	109	121
NAMD		100	100	112	149
povray31		168	144	262	512

Figure 4.11: Number of Loads Hoisted or Sunk

We elide zeros from the table.

Benchmark	local	steens-fi	steens-fs	anders	ds-aa
SPEC CINT 2000					
181.mcf		22	22	22	22
256.bzip2	5	23	23	23	31
164.gzip	2	10	10	10	18
175.vpr	7	546	546	654	792
197.parser	3	46	46	63	73
186.crafty	7	58	58	59	179
300.twolf	16	581	581	626	878
255.vortex	29	159	159	280	476
254.gap	1	34	34	53	271
252.eon	8	73	73	106	543
253.perlbmk	22	41	41	63	52
176.gcc	18	369	369	430	585
SPEC CFP 2000					
179.art		98	98	98	98
183.equake	3	77	77	77	77
171.swim	1	4	4	4	32
172.mgrid	1	6	6	6	29
168.wupwise	30	37	37	40	65
173.applu	1	4	4	25	17
188.ammp		22	22	69	82
177.mesa	2	94	94	91	127
SPEC CINT 1995					
129.compress					1
130.li					16
124.m88ksim	1	11	11	13	36
132.ijpeg		83	85	95	211
099.go	29	34	34	34	88
134.perl	6	54	54	82	62
147.vortex	29	159	159	280	476
126.gcc	11	369	369	428	582
SPEC CFP 1995					
102.swim	1	4	4	1	32
101.tomcatv	2	7	7	7	17
107.mgrid	1	6	6	10	24
145.fpppp	11	28	28	30	54
104.hydro2d	3	19	19	56	63
110.applu	1	4	4	25	17
103.su2cor	3	13	13	37	240
146.wave5	3	6	6	62	78
Other Programs					
fpgrowth		13	14	25	32
boxed-sim		56	56	165	177
NAMD	1	102	102	114	154
povray31	14	203	179	362	826

Figure 4.12: Number of Instructions Hoisted or Sunk

We elide zeros from the table.

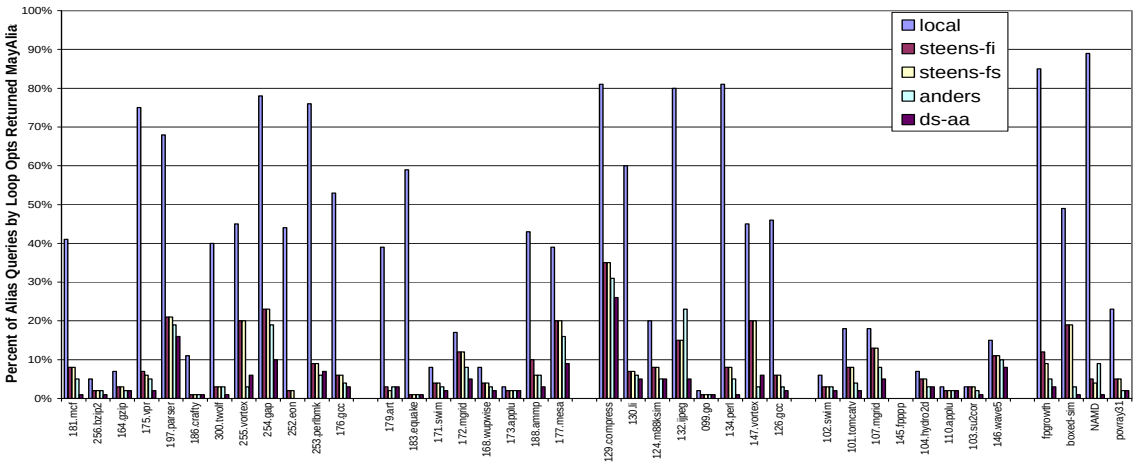


Figure 4.13: Percent of LICM Alias Queries Returned "May Alias"

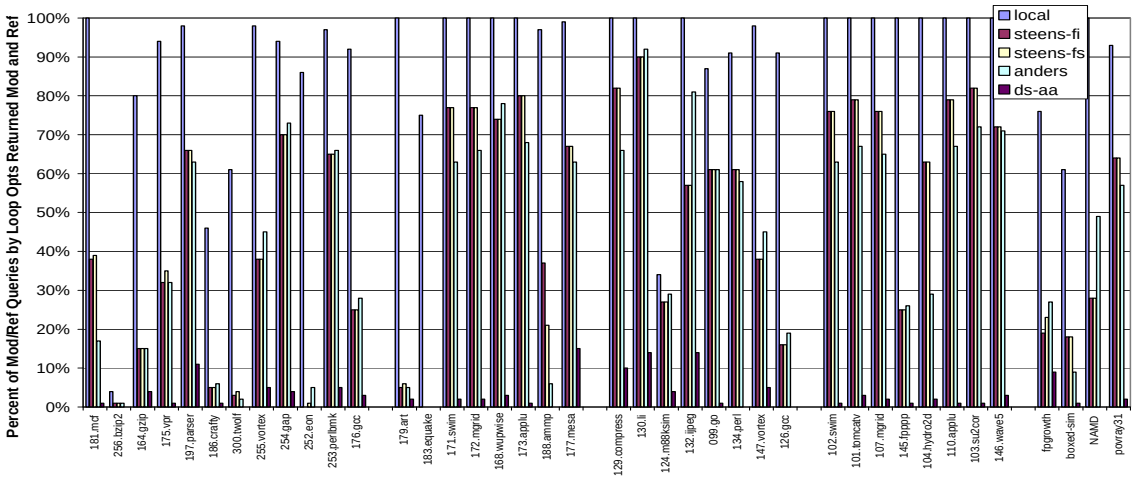


Figure 4.14: Percent of LICM Mod/Ref Query Responses Returned "Mod and Ref"

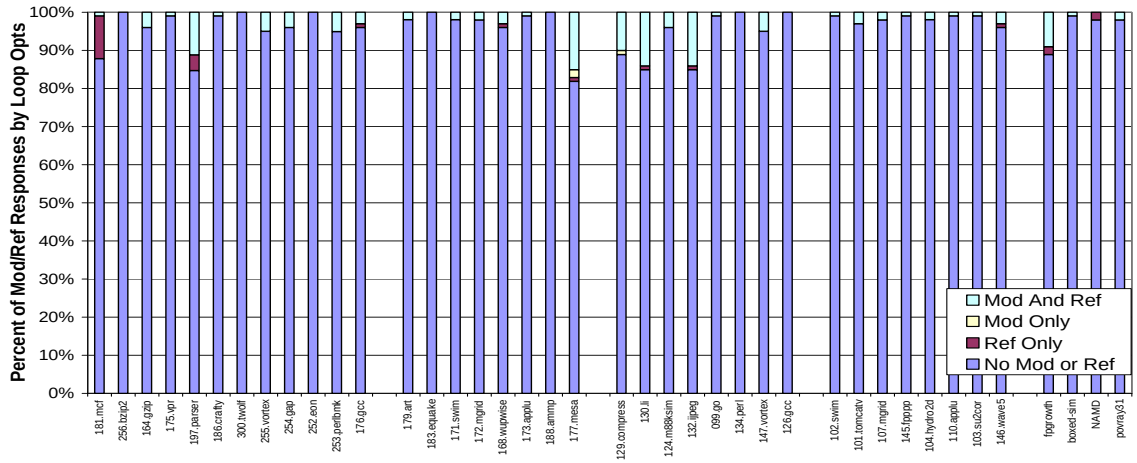


Figure 4.15: DSA LICM Mod/Ref Query Responses Breakdown

because of phase ordering issues).

The table generally agrees with the metrics gathered in Section 4.4 for the AA-EVAL client. In particular, the **local** analysis is able to perform few transformations (having already been used earlier in compilation). **steens-fi** and **steens-fs** enable far more optimization possibilities than **local**, and **steens-fs** performs a few transformation more than **steens-fi** in some cases (e.g. more locations are promoted to registers in povray31). The **anders** analysis provides a far more precise result than any of the previous results, and **ds-aa** enables more optimizations than **anders** in most cases (sometimes far more, e.g. 252.eon, 254.gap, 103.su2cor, sometimes slightly less, e.g. 253.perlbmk, 173.applu).

4.5.2 Alias and Mod/Ref Queries

Figures 4.13, 4.14, and 4.15 contain information about how the different analyses responded to the queries made by the scalar loop optimizer. Figure 4.13 shows the number of MayAlias responses returned by the analysis for load/store and store/store dependence checks (compare with Figure 4.3). Figure 4.14 shows the number of call/location mod/ref queries that are returned as "Mod and Ref" (compare to Figure 4.4). Finally, Figure 4.15 shows the breakdown of mod/ref queries as returned by DSA (compare with Figure 4.8). We elide the other charts, as they are either irrelevant ("ref only" and "no mod ref" responses does not affect this transformation), or contain very little important data (e.g., the extremely sparse "mod only" chart).

The numbers in these charts are often correlated, but sometimes quite different than the precision numbers provided in Section 4.4. Overall, the "success rate" of the queries is much higher for this client than for the synthetic client (for example, the percentage of may alias responses is generally lower). There are four potential reasons for this:

- The loop optimizer performs several other legality checks before it queries alias analysis. In particular, if the pointer is not loop invariant, no alias queries will be performed (and obviously it will never make queries for values outside of loops). These checks can significantly bias the queries made of the alias analysis (for example, making queries of globals more frequent). It is possible (and somewhat likely) that the alias analyses are better at disambiguating cases that pass the initial legality tests.

- The loop optimizer does not perform all possible queries inside of a loop body. In particular, if it is attempting to hoist a load, it will perform mod/ref queries of the loaded location against all of the potentially modifying instructions in the loop body until it hits one that might modify the memory location. Analysis counts will differ if the last operation in the loop modifies the location compared to if the first operation in the loop modifies the location.

- Hoisting one instruction may make other dependent instructions hoistable (for example "T = ***p"). This can bias query percentages in strange ways: for example, a precise analysis that hoists a large number of operations will cause the loop optimizer to make many more queries than it would of an imprecise analysis. This makes it almost impossible to compare bars for different analyses on the same benchmark.

- The loop optimizer occasionally makes duplicate queries. Because we count the raw number of executed transactions between the alias analysis and the client, these duplicate queries can also bias the results.

Because there are so many issues that can bias the results in strange ways, both comparing this set of data to the AA-EVAL data, and even comparing the bars for one analysis to the bars for another, we don't feel that comparing success rate is a good idea. Comparing number of transformations enabled and performed is a more important and accurate metric.

4.6 Observations and Conclusions

- **DSA can support traditional alias and mod/ref clients in the same framework that it uses to support the macroscopic clients described throughout this thesis.** Our primary goal with DSA is to provide an extremely fast analysis framework that captures the important properties of memory in the program. Because all of the algorithms in this thesis depend on an analysis like DSA, DSA's precision and generality are very important for this work.

- **A scalable context-sensitive unification-based analysis can be more precise and more useful than a non-context-sensitive subset-based algorithm while requiring**

approximately the same analysis time. One of the main results of DSA is to demonstrate that a fully context-sensitive algorithm can be both extremely fast and scalable. Given this extremely fast analysis, a compiler engineer can choose either a non-context-sensitive subset-based approach or a context-sensitive unification-based approach to achieve good precision with reasonable compile-time cost (while scalable cloning-based context-sensitive subset-based are available [140], in practice they are orders of magnitude slower than DSA). The advantage of using a DSA-like approach is that it enables the full suite of macroscopic approaches described in this thesis.

- **mod/ref information is an extremely useful property which cannot be aggressively captured by a non-context sensitive algorithm**. If non-context-sensitive pointer analysis algorithms are used as the main analysis, they must be followed by context-sensitive mod/ref summary algorithms. We show that simple redundancy elimination optimizations are greatly enhanced by context-sensitive mod/ref information, and describe how DSA computes this information as part of its analysis.

- **Field-Sensitivity is a straight-forward extension of Steensgaard's algorithm which can improve precision in some cases**. When implemented with speculative type information (as DSA/ds-aa and steens-fs do), there is very little additional compile-time cost to preserving field information. This information gives a marginal improvement in alias analysis precision (for a non-context-sensitive unification-based algorithm), and can be used by more aggressive analyses (like DSA) for higher level analyses and transformations (such as macroscopic techniques). We believe that the value of field-sensitivity has been largely ignored, except as a way of increasing points-to precision.

Note that our evaluation specifically does not evaluate, compare against, or draw conclusions about subset-based heap-cloning context-sensitive pointer analysis algorithms. In particular, we explicitly do not draw any conclusions on the effect of adding context sensitivity to a subset-based analysis. Others have shown that this can substantially improve alias analysis precision, though at potentially very large additional analysis cost (e.g., 50-100x in some cases in recent work [140, 103]) over the cost of a non-context-sensitive subset-based approach. This context-sensitivity changes the

analysis time requirements from seconds to minutes (or hours depending on the size of the code) for medium sized programs (e.g. 100,000 lines of C code).

In our exerience, we find that use of unification provides a simple solution to hard problems that heap-cloning context-sensitive algorithms face, and we have not witnessed significant imprecision in heap analysis that would be prevented by the use of a subset-based approach. Intuitively, this is because there is a limited amount of information that any flow-insensitive algorithm can say about recursive data structures, and in practice, field-sensitive subset-based analysis and field-sensitive unification-based analysis can prove the same things. To get more information, flow-sensitive techniques with strong updates (e.g. shape analysis) must be used, providing an additional level of information.

In contrast, we frequently witness pessimization of global variables and (occasionally) stack object analysis precision due to the use of unification. Specifically, any time a program uses a pointer to two globals, a unification-based analysis will not be able to *ever* distinguish between the two. This experience leads us to propose the following conjecture, which we consider to be an open research question:

Conjecture 4.6.1 *Subset-based pointer analysis does not provide any substantial precision advantage (over unification-based analysis) for heap allocated memory objects in a fully context-sensitive pointer analysis.*

Studies have shown that context-sensitivity improves the precision of both unification based (this work, [41, 92], etc) and subset-based ([140, 103]) algorithms. We believe that the precision difference between a heap-cloning context-sensitive subset-based analysis and a similar unification-based analysis is mostly due to differences in global and stack object analysis precision, not due to heap object analysis precision. This belief leads us to propose the following conjecture:

Conjecture 4.6.2 *A heap-cloning context-sensitive alias analysis algorithm that uses unification-based analysis for heap objects and subset-based analysis for stack objects and globals may be an important compromise that provides precision close to the leading fully context-sensitive subset-based approaches and analysis times that are significantly better (perhaps approaching the speed of DSA).*

We think that investigation and evaluation of this hybrid algorithm is an important open research problem, but leave investigation to future work.

Chapter 5

Automatic Pool Allocation

One of the most important tasks for modern compilers and runtime systems is the management of memory usage in programs, including safety checking, optimization, and storage management. Unfortunately, while compiler techniques for analyzing and controlling memory access patterns for dense arrays has proven very effective, techniques for dealing with pointer-based data structures are much weaker. A key difference between the two is that compilers have precise knowledge of the runtime layout of arrays in memory, whereas they have much less information about complex data structures allocated on the heap. In such (pointer-based) data structures, both the relative layout of *distinct data structures* in memory (which affects working set sizes) and the layout of nodes *within a single data structure* (which affects memory traversal patterns) are difficult to predict. One direct consequence is that irregular memory traversal patterns often have worse performance, both because of poor spatial locality and because techniques like hardware stride prefetching are not effective. A potentially more far-reaching consequence is that many compiler techniques (e.g., software prefetching, data layout transformations, and safety analysis) are either less effective or not applicable to complex data structures.

This chapter describes **Automatic Pool Allocation**, a transformation framework for arbitrary imperative programs that *segregates distinct instances of heap data structures into seperate memory pools*, and allows heuristics to be used to partially control the internal layout of those data structures. For example, each distinct instance of a list, tree, or graph identified by the compiler would be allocated to a separate pool. Our transformation uses the output of a context-sensitive, field-sensitive points-to analysis (DSA)[1] to distinguish *disjoint instances of logical data structures*

[1]Less aggressive pointer analyses can also be used but may not distinguish data structure instances or may give

in a program, and identify the locations at which nodes of those data structures are created, accessed, and destroyed. This gives the compiler the information needed to segregate individual data structure instances and to better control their internal layout.

The Automatic Pool Allocation algorithm supports arbitrary C and C++ programs, including programs with function pointers and/or virtual functions, recursion, varargs functions, non-type-safe memory accesses (e.g., via pointer casts and unions), setjmp/longjmp, and exceptions. One of the key strengths of the algorithm is a simple strategy for correctly handling indirect calls, which is difficult because different functions called via a function pointer may have different allocation and deallocation behavior and because (in C or C++) may even have different signatures. The algorithm solves these complex issues via a relatively simple graph transformation phase, while keeping the code transformation process essentially unchanged. The transformation works correctly for incomplete programs, by only pool allocating memory that does not escape the scope of analysis.

The Automatic Pool Allocation provides several novel features, compared to previous work on region inference. In particular, it is the first approach that builds on a scalable context-sensitive pointer analysis, works with non-type-safe programs, supports functions with varargs, allows arbitrary function pointer handling, etc. In addition, Automatic Pool Allocation is the first approach which is designed both improve program performance (through better locality), and provide a framework for subsequent compiler optimizations.

Automatic Pool Allocation can directly improve program performance in several ways. First, since programs typically traverse and process only one or a few data structures at a time, segregating logical data structures reduces the memory working sets of programs and potentially improving both cache and TLB performance. Second, in certain cases, the allocation order within each data structure pool will match the subsequent traversal order (e.g., if a tree is created and then processed in preorder), improving spatial locality and (if objects are smaller than a cache line) temporal locality. Intuitively, both benefits arise because the layout of individual data structures is unaffected by intervening allocations for other data structures, and less likely to be scattered around in the heap. Third, in some cases, the traversal order may even become a simple linear stride, allowing more effective hardware prefetching than before. Note that Automatic Pool Allocation

less precise information about their internal structure.

can also potentially hurt performance in two ways: by separating data that are frequently accessed together and by allocating nearly-empty pages to small pools (some of the optimizations described in Chapter 6 address this).

At the end of Chapter 6, we present an experimental study of the performance impact of Automatic Pool Allocation to show the execution time and locality effects of the transformation. We find that several programs speed up by 10-20%, two by about 2x and two by more than 10x. Other programs are unaffected, and importantly, none are hurt substantially by the transformation. We also graphically show how segregation of data structures in memory can provide a 10x performance improvement in some cases.

This chapter starts by describing the running example we use and gives a high-level overview of the transformation (Section 5.1). Next it describes the full algorithm in detail in Section 5.2, its complexity in Section 5.3, and several simple (but important) refinements in Section 5.4. Section 5.5 evaluates the compile time and static pool allocation statistics on a broad range of pointer intensive programs, and Section 5.6 contrasts Automatic Pool Allocation with prior work. Finally, Section 5.7 summarizes the contributions of the pool allocation algorithm described in this chapter.

5.1 The Transformation Overview and Example

The pool allocation transformation operates on a program containing calls to `malloc` and `free`, and transforms the program to use a pool allocation library, described below. The algorithm uses a points-to graph and call graph, both of which are computed by DSA in our implementation. The transformation is a framework which has several optional refinements. In this section, we present a "basic" version of the transformation in which all heap objects are allocated in pools (i.e., none are allocated directly via `malloc`) and every DS node generates a separate static pool (explained below). All steps of the algorithm consider only those DS nodes with $\mathbf{H} \in M$ ("**H** nodes") as candidates for allocating to pools. Because the DSGraphs identified by DSA identify disjoint memory objects, this transformation automatically segregates such data structure instances in the heap. In the next section, we discuss additional refinements to this basic approach.

5.1.1 Pool Allocator Runtime Library

Figure 5.1 shows the interface to the runtime library. Pools are identified by a pool descriptor of type `Pool`. The functions `poolalloc`, `poolfree`, and `poolrealloc` allocate, deallocate, and resize memory in a pool. The `poolcreate` function initializes a pool descriptor for an empty pool, with an optional size hint (providing a fast path for a commonly allocated size) and an alignment required for the pool (this defaults to 8, as in many standard malloc libraries). `pooldestroy` releases all pool memory to the system heap. The last three functions (with suffix "`_bp`") are variants that use a fast "bump pointer" allocation method, described in Section 6.1.3.

```
void poolcreate(Pool* PD, uint Size, uint Align)
    Initialize a pool descriptor.
void pooldestroy(Pool* PD)
    Release pool memory and destroy pool descriptor.
void* poolalloc(Pool* PD, uint numBytes)
    Allocate an object of numBytes bytes.
void poolfree (Pool* PD, void* ptr)
    Mark the object pointed to by ptr as free.
void* poolrealloc(Pool* PD, void* ptr, uint numBytes)
    Resize an object to numBytes bytes.

void poolinit_bp(Pool *PD, uint Align)
    Initialize a bump-pointer pool descriptor.
void *poolalloc_bp(Pool *PD, uint NumBytes)
    Allocate memory from a bump-pointer pool.
void pooldestroy_bp(Pool *PD)
    Release a bump-pointer pool.
```

Figure 5.1: Interface to the Pool Allocator Runtime Library

The library internally obtains memory from the system heap in blocks of one or more pages at a time using `malloc` (doubling the size each time). We implemented multiple allocation algorithms but the version used here is a general free-list-based allocator with coalescing of adjacent free objects. It maintains a four-byte header per object to record object size. The default alignment of objects (e.g., 4- or 8-byte) can be chosen on a per-pool basis, for reasons described in Section 6.1.4. The pool library is general in the sense that it does not require all allocations from a pool to be the same size.

```
struct list { list *Next; int *Data; };

list* createnode(int *Data) {
  list *New = malloc(sizeof(list));
  New->Data = Data;
  return New;
}
void splitclone(list *L, list **R1,
                list **R2) {
  if (L == 0) { *R1 = *R2 = 0; return; }
  if (some_predicate(L->Data)) {
    *R1 = createnode(L->Data);
    splitclone(L->Next, &(*R1)->Next, R2);

  } else {
    *R2 = createnode(L->Data);
    splitclone(L->Next, R1, &(*R2)->Next);

  }
}
int processlist(list* L) {
  list *A, *B, *tmp;

  // Clone L, splitting into list A and B.
  splitclone(L, &A, &B);
  processPortion(A);  // Process first list
  processPortion(B);  // process second list

  // free A list
  while (A) {
    tmp = A->Next; free(A); A = tmp;
  }
  // free B list
  while (B) {
    tmp = B->Next; free(B); B = tmp;
  }

}
```

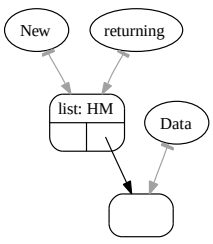
(a) Input C program manipulating linked lists

```
struct list { list *Next; int *Data; };

list* createnode(Pool *PD, int *Data) {
  list *New = poolalloc(PD, sizeof(list));
  New->Data = Data;
  return New;
}
void splitclone(Pool *PD1, Pool *PD2,
                list *L, list **R1, list **R2) {
  if (L == 0) { *R1 = *R2 = 0; return; }
  if (some_predicate(L->Data)) {
    *R1 = createnode(PD1, L->Data);
    splitclone(PD1, PD2,
               L->Next, &(*R1)->Next, R2);
  } else {
    *R2 = createnode(PD2, L->Data);
    splitclone(PD1, PD2,
               L->Next, R1, &(*R2)->Next);
  }
}
int processlist(list* L) {
  list *A, *B, *tmp;
  Pool PD1, PD2;  // initialize pools
  poolcreate(&PD1); poolcreate(&PD2);
  splitclone(&PD1, &PD2, L, &A, &B);
  processPortion(A);  // Process first list
  processPortion(B);  // process second list

  // this loop is eventually eliminated
  while (A) {
    tmp = A->Next; poolfree(&PD1, A); A = tmp;
  }
  // this loop is eventually eliminated
  while (B) {
    tmp = B->Next; poolfree(&PD2, B); B = tmp;
  }
  pooldestroy(&PD1); pooldestroy(&PD2);
}
```

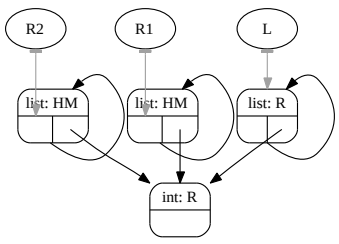
(b) C code after the basic pool allocation transformation

Figure 5.2: Example illustrating the Pool Allocation Transformation

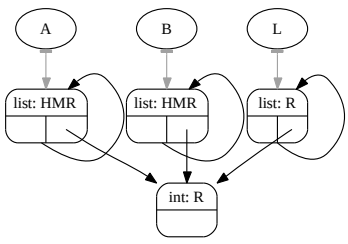
'processlist' copies a list into two disjoint lists (based on some predicate), processes each, then frees them. After basic pool allocation, the new lists are put in separate pools (PD1 and PD2) which are each contiguous in memory. After subsequent optimization described in Chapter 6, the calls to `poolfree` *and the loops containing them are removed because* `pooldestroy` *atomically frees all pool memory.*



(a) DS Graph for createnode



(b) DS Graph for splitclone



(c) DS Graph for processlist

Figure 5.3: BU DSGraphs for functions in Figure 5.2 (a)

5.1.2 Overview Using an Example

The basic pool allocation transformation is illustrated for the example program in Figure 5.2(b), which shows the results of our basic transformation in C syntax. The incoming list L and the two new lists have each been allocated to distinct pools (the pool for L is not passed in and so not shown; the new lists use pools PD1 and PD2). The list nodes for A and B will be segregated in the heap, unlike the original program where they will be laid out in some unpredictable fashion (and possibly interleaved) in memory. The items in each pool are explicitly deallocated and the pools are destroyed within `processList` when the data they contain is no longer live.

We can use this example to explain the basic steps of the transformation. The DS graphs are shown in Figure 5.3. First, we use each function's DS graph to determine which **H** nodes are accessible outside their respective functions, i.e., "escape" to the caller. The **H** nodes in `createnode` and `splitclone` do escape, because they are reachable from a returned pointer and a formal argument, respectively. The two in `processlist` (`A` and `B`) do not. The latter are candidates for new pools in `processlist`.

The transformation phase inserts code to create and destroy the pool descriptors for `A` (`PD1`) and `B` (`PD2`) in `processlist` (see Figure 5.2(b)). It adds pool descriptor arguments for every **H** node that escapes its function, i.e., for nodes pointed to by `R1` and `R2` in `splitclone` and the node pointed to by `New` in `createNode`. It rewrites the calls to `malloc` and `free` with calls to `poolalloc` and `poolfree`, passing appropriate pool descriptors as arguments. Finally, it rewrites other calls to (e.g., the calls to `splitclone` and `createnode`) to pass any necessary pool descriptor pointers as arguments. At this point, the basic transformation is complete.

Further refinements of the transformation move the `pooldestroy` for `PD1` as early as possible within the function `processlist`, and then eliminate the calls to free items in the two lists (since these items will be released by pooldestroy before any new allocations from any pool) and hence the loop enclosing those calls to free. The final resulting code (Figure 6.1) puts each linked list into a separate pool on the heap, made the list objects of each list contiguous in memory, and reclaims all the memory for each list at once instead of freeing items individually. In the example, the list nodes are placed in dynamic allocation order within their pool.

5.2 The Core Pool Allocation Transformation

The pool allocation transformation consists of two main parts: an analysis to find which functions host pools in the program (Section 5.2.1), and the code transformation that rewrites the program to allocate and free memory from these pools. To simplify presentation, we first present a version of the algorithm that does not support indirect function calls (Section 5.2.2), then extend the basic algorithm to support indirect calls (Section 5.2.3).

5.2.1 Analysis: Finding Pool Descriptors for each H Node

The analysis phase identifies which pool descriptors must be available in each function, determines where they must be created and destroyed, and assigns pool descriptors to DS nodes. We use the term *static pool* to refer to a single `poolcreate` statement in the generated code. By definition, $\mathbf{H} \in M$ for a node if the objects of that node are returned by `malloc` or passed into `free` by the current function or any of its callees, since we assume a Bottom-up DS graph. These identify exactly those nodes for which a pool descriptor must be *available* in the current function.

Automatic Pool Allocation computes a map (pdmap) identifying the pool descriptor corresponding to each DS node with $\mathbf{H} \in M$. We initially restrict pdmap to be a one-to-one mapping from DS nodes to pool descriptor variables; Section 6.2 extends pdmap to allow a many-to-one mapping. We must handle two cases: 1) the pool escapes the current function and 2) the pool lifetime is bound by the function. In the first case, we add a pool descriptor argument to the function, in the second, we create a descriptor on the stack for the function and call `poolcreate`/`pooldestroy`. These two cases are differentiated by the "escapes" property for the DS node.

The "escapes" property is determined by a simple escape analysis [14] on the bottom-up DS graphs, implemented as a depth-first traversal. In particular, a node escapes iff 1) a pointer to the node is returned by the function (e.g. `createnode`) 2) the node is pointed to by a formal argument (e.g. the R1 node in `splitclone`) 3) the node is pointed to by global variable and the current function is not main, or 4) (inductively) an escaping node points to the node.

A subtle point is that any node that does not escape a function will be unaffected by callers of the function, since the objects at such a node are not reachable (in fact, may not exist) before the current function is called or after it returns. *This explains why it is safe to use a BU graph*

for pool allocation: Even though the BU graph does not reflect any aliases induced by callers, the non-escaping nodes are correctly identifiable and all information about them is complete, including their type τ, incoming points-to edges, and flags. In fact, in DSA, the escapes property is explicitly computed and all non-escaping nodes are marked using a "**C**"omplete flag (See Section 3.1.1). It can be computed easily using the above definition by any context-sensitive algorithm that has similar points-to graphs.

5.2.2 The Simple Transformation (No Indirect Calls)

Figure 5.4 shows the pseudocode for a basic version of the Automatic Pool Allocation transformation, which does not handle indirect function calls. The algorithm makes two passes over the functions in the program in arbitrary order. The first (lines 1–11) adds arguments to functions, creates local pool descriptors, and builds the pdmap. The second (lines 12–20) rewrites the bodies of functions using pdmap.

```
   basicpoolallocate(program P)
1  ∀F ∈ functions(P)
2    dsgraph G =DSGraphForFunction(F)
3    ∀n ∈ nodes(G)              // Find pooldesc for heap nodes
4      if (H ∈ n.M)
5        if (escapes(n))        // If node escapes fn
6          Pool* a = AddPoolDescArgument(F, n)
7          pdmap(n) = a         // Remember pooldesc
8          argnodes(F) = argnodes(F) ∪ {n}
9        else                   // Node is local to fn
10         Pool* pd = AddInitAndDestroyLocalPool(F, n)
11         pdmap(n) = pd
12 ∀F ∈ functions(P)
13   ∀I ∈ instructions(F)       // Rewrite function
14     if (I isa 'ptr = malloc(size)')
15       replace I with 'poolalloc(pdmap(N(ptr)), size)'
16     else if (I isa 'free(ptr)')
17       replace I with 'poolfree(pdmap(N(ptr)), ptr)'
18     else if (I isa 'call Callee(args)')
19       ∀n ∈ argnodes(Callee)
20         addCallArgument(pdmap(NodeInCaller(F, I, n)))
```

Figure 5.4: Pseudo code for basic algorithm

For each node that needs a pool in the function, the algorithm either adds a pool descriptor argument (if the DS node escapes) or it allocates a pool descriptor on the stack. Non-escaping pools are initialized (using `poolcreate`) on entry to the function and destroyed (`pooldestroy`) at every exit of the function (these placement choices are improved in Section 5.4.2). Because the DS

node does not escape the function, we are guaranteed that any memory allocated from that pool can never be accessed outside of the current function, i.e., it is safe to destroy the pool, even if some memory was not deallocated by the original program. Note that this may actually eliminate some memory leaks in the program!

In the second pass (lines 12–20), the algorithm replaces calls to `malloc()` and `free()`[2] with calls to `poolalloc` and `poolfree`. We pass the appropriate pool descriptor pointer using the pdmap information saved by the first pass. Since the DS node must have an **H** flag, a pool descriptor is guaranteed to be available in the map.

Calls to functions other than `malloc` or `free` must pass additional pool descriptor arguments for memory that escapes from them. Because the BU Graph of the callee reflects all accessed memory objects of all transitive callees, any heap objects allocated by a callee will be represented by an **H** node in the caller graph (this is true even for recursive functions like `splitclone`). This property guarantees that a caller will have all of the pool descriptors that any callee will ever need.

A key primitive computable from DS graphs is a mapping, $NodeInCaller(F, C, n)$. For a call instruction, C, in a function F, if n is a DS node in any possible callee at that call site, then $n' = NodeInCaller(F, C, n)$ identifies the node in the DS graph of F corresponding to node n due to side-effects of the call C (i.e., n' includes the memory objects of node n visible in F due to this call). The mapping is computed in a single linear-time traversal over matching paths in the caller and callee graphs, starting from matching pairs of actual and formal nodes, matching pairs of global variable nodes, and the return value nodes in the two graphs if any. If n escapes from the callee, then the matching node n' is guaranteed to exist in the caller's BU graph (due to the bottom-up inlining process used to construct the BU graphs), and is unique because the DS graphs are unification-based (see Section 3.2.3).

Identifying which pool of the caller (F) to pass for callee pool arguments at call instruction I is now straightforward: for each callee node n that needs an argument pool descriptor, we pass the pool descriptor for the node $NodeInCaller(F, I, n)$ in the caller's DS graph. We record the set of nodes ("argnodes") that must be passed into each function, in the first pass.

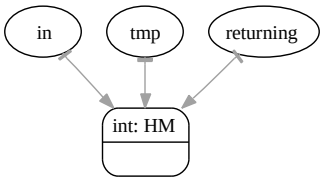
[2]Note that "malloc wrappers" (like `calloc`, `operator new`, `strdup`, etc) do not need special support from the pool allocator. Their bodies are simply linked into the program and treated as if they were a user function, getting new pool descriptor arguments to indicate which pool to allocate from.

```
int* func1(int* in) {
  *in = 1;
  return in;
}
int* func2(int* in) {
  free(in);
  in = malloc(sizeof(int));
  *in = 2;
  return in;
}
int caller(int X) {

  int* (*fp)(int*) = (X > 1)?func1:func2;
  int *p = malloc(sizeof(int));
  int *q = fp(p);

  return *q;
}
```

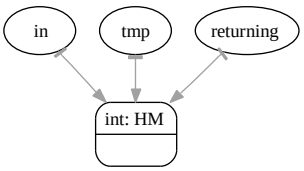
(a) Input C program with indirect function call

```
int* func1(Pool* P, int* in) {
   *in = 1;
   return in;
}
int* func2(Pool* P, int* in) {
   poolfree(P, in);
   in = poolalloc(P, sizeof(int));
   *in = 2;
   return in;
}
int caller(int X) {
  Pool PD1;  poolcreate(&PD1, ...);
  int* (*fp)(int*) = (X > 1)?func1:func2;
  int *p = poolalloc(PD1, sizeof(int));
  int *q = fp(PD1, p);
  pooldestroy(&PD1);
  return *q;
}
```

(b) C code after pool allocation



(c) Merged EBU Graph for `func1` and `func2`



(d): EBU Graph for `caller`

Figure 5.5: Pool Allocation Example with Function Pointers
Though `func1` and `func2` are called at the same call site, only one needs a pool descriptor. The algorithm puts them in a single equivalence class, merges their DS graphs, adds a pool argument to both functions.

Variable-argument functions do not need any special treatment in the transformation because of their representation in the BU graphs computed by DSA. In particular, the DS graph nodes for all pointer-compatible arguments passed via the "`...`" mechanism (i.e., received via `va_arg`) are merged so that they are represented by a single DS node in the caller and callee. If the DS node pointed to by this argument node has $\mathbf{H} \in M$, a single pool argument is added to the function. At every call site of this function, the nodes for the actual argument (corresponding to the merged formals) will also have been merged, and the pool corresponding to this node will be found by $NodeInCaller(F, I, n)$ and passed in as the pool argument. Note that explicit arguments before the `...` are not merged and can have distinct pools.

5.2.3 Passing Descriptors for Indirect Function Calls

Indirect function calls make it much more complex to pass correct pool descriptor arguments to each function. There are multiple difficulties. First, different functions called via a function pointer at the same call site may require different sets of pools. Figure 5.5 shows a simple example where `func1` needs no pools but `func2` needs one pool, and both are called at the same site. Second, different indirect call sites can have different but overlapping sets of callees, e.g., $\{F_1, F_2\}$ and $\{F_2, F_3\}$ at two different call sites. In order to avoid cloning F_2 into two versions, we must pass the same pool arguments to all three functions F_1, F_2 and F_3. This raises a third major problem: because the call graph says that F_3 is not a callee at the first call-site, its DS graph was never inlined into that of the caller at that call-site. This means that the matching of nodes between caller and callee graphs, which is essential for passing pool descriptors, may be undefined: $NodeInCaller(F, C, n)$ may not exist for all escaping n. Programs that violate the type signatures of functions at call sites (not uncommon in C code) exacerbate all three problems because any attempt to match pool arguments explicitly for different callees must account for mismatches between the actual and formals for each possible callee.

Our solution is composed of two key principles, described below, and shown in pseudocode in Figure 5.6. The first principle is to partition into equivalence classes so that all potentially callees at an indirect call site are in the same class. We then treat *all* functions in the same equivalence class as potential callees for that call site. For example, `func1` and `func2` in the example figure are put into the same class, and so are F_1, F_2 and F_3 in the example above. Lines 1-2 uses the call graph to partition all the functions of the program into disjoint equivalence classes in this manner.

The second principle is to simplify matching nodes between different callees at a call site with the nodes of the caller by merging the graphs of all functions in an equivalence class, and then updating the caller graphs to be consistent with the merged callee graphs. Merging the graphs ensures that an identical set of pool descriptor formal arguments will be inferred for all functions in the class. Updating the caller graphs to be consistent with the callee graphs (as explained below) ensures that the third problem above — finding matching nodes between callee and caller — is always possible.

In the example, the algorithm merges the DS graphs of `func1` and `func2` into the common graph

shown in Figure 5.5(c), and uses this common graph to transform both functions. This results in a matching set of pool arguments for both functions, even though the pool will be unused in `func1`. This common graph is merged into the caller, resulting in the graph shown in Figure 5.5(d). Using this graph, one descriptor is passed to both functions at the call site.

The implementation of these graph merging and inlining steps (lines 3-8 of Figure 5.6) use two primitive DSA operations – merging two graphs and performing a bottom-up inlining pass on strongly-connected components (SCCs) of the call graph. To merge the graphs of two functions in an equivalence class (lines 3-4), we copy one graph into the other, then unify corresponding formal argument nodes (ignoring any extra nodes in one of the graphs if the formal argument lists do not match), global nodes, and the return value node of each graph. Unifying nodes causes recursive merging and can potentially cause loss of some type information if merged nodes have incompatible types.

Finally, we perform a bottom-up "inlining" pass on the strongly connected components (SCCs) of the call graph, inlining merged graphs of the callees into their callers. This simply requires repeating the bottom-up inlining pass of the DSA algorithm (starting with the merged equivalence-class graphs of each function). This step is described in detail in Section 3.2.3.

We call the resulting DS graphs the EBU ("equivalence bottom-up") graphs. The EBU graph is more conservative than the original DS graph because functions known not to be called at a call-site may be merged into the caller along with those that are (because they are in the same equivalence class). Such cases do not arise often in practice, and the merging of equivalence class graphs greatly simplifies the overall transformation algorithm by solving the above three problems with a uniform strategy based on existing DS graph primitives.

```
   completepoolallocate(program P)
1    ∀cs ∈ callsites(P)                    // Build equivalence classes
2      unify equivclasses(callees(cs))
3    ∀ec ∈ equivclasses(functions(P))      // Build graph for each class
4      ECGraph(ec) = mergeGraphs(DSGraphs(members(ec)))
5    ∀scc ∈ tarjansccfinder(callgraph(P))
6      ECGraph(scc) = mergeGraphs(ECGraphs(functions(scc)))
7      ∀cs ∈ callsites(scc)                // Inline callees into caller
8        ECGraph(scc) = mergeGraph(cs, ECGraph(callees(cs)))
9    basicpoolallocate(P)
```

Figure 5.6: Pseudo code for complete pool allocator

Given the EBU graphs for a program, the pool allocator is now guaranteed to have all of the

pool descriptors required at an indirect call site for any of the potential callees of the call site, allowing it to apply the `basicpoolallocate` algorithm safely. Note that lines 17-19 simply have to use the common graph for all callees even though there may now be multiple callers for the call at line 17.

5.3 Algorithmic Complexity

The first phase of the basic pool allocation transformation itself (lines 1–10 in Figure 5.4) is linear in the total number of DS graph nodes because all escaping nodes can be found with a single traversal each DS graph, and the remaining steps are trivially linear. We have found empirically that the total number of DS graph nodes scales essentially linearly with program size, a consequence of using a unification based algorithm (although field-sensitivity can cause the number of DS graph nodes to grow more quickly with program size, this occurs only in pathological cases because of the node merging effect of unification, see Section 3.2.6). The second phase (lines 11–20) is linear in the number of memory allocation and deallocation operations and call sites in the program, plus the number of pool arguments added which is itself linear in the number of DS graph nodes.

The EBU graph merging phase, which precedes the transformation, has three main steps (lines 1–8 of Figure 5.6). The first step (lines 1–2) is $O(c + f\alpha(f))$ if c and f are the number of edges and functions in the program call graph and α is the inverse Ackerman's function. The `mergeGraphs` operation is equivalent to the node merging performed in any unification based algorithm, and requires $O(k\alpha(k))$ time for graphs of size k. The first merging phase (lines 3–4) merges each function's graphs no more than once into it's equivalence-class graph, so it's complexity is similar. The second merging phase is equivalent to a subset of the BU phase of Data Structure Analysis and has the same complexity as that phase, viz., $\Theta(n\alpha(n) + k\alpha(k)c)$, if n, k and c are denote the total number of instructions, the maximum size of a DS graph for a single procedure, and the number of edges in the call graph. In practice, k is very small, typically on the order of a hundred nodes or less, even in programs of over 100K lines of C code (See Section 3.2.6).

Overall, the complexity of the pool allocation algorithm is close to linear in the size of the input DS graphs, and (empirically) close to linear in overall program size. In practice, it is extremely fast, as we show experimentally in Section 5.5.3.

5.4 Simple Pool Allocation Refinements

This section describes two simple refinements to the pool allocation algorithm which permit it to generate far more efficient code in some cases.

5.4.1 Argument Passing for Global Pools

Efficient handling of pools reachable from global variables is optional for functionality, but absolutely required for performance. Note that the pool for any node reachable from a global must be created in `main` because the data may be live throughout the lifetime of the program. The major problem this causes is that such a pool would have to passed down through many layers of function calls to be available at each function that actually allocates or frees data in the pool. In practice, programs which have many heap nodes reachable from globals may get thousands of arguments added to the program.

The solution to this problem is simple: we create a global variable to hold the pool descriptor for each heap node reachable from a global and use this in place of arguments passed into a function where possible. In practice, this refinement greatly reduces the number of pool arguments that must be passed to functions in some C programs. Most importantly, it ensures that the only pool arguments that must be passed to a function are for pointers passed in as formal arguments to that function (or nodes reachable from such pointers), ensuring that the number of pool arguments is roughly proportional to the number of formal pointer arguments in the original function.

Finally, using global variables as pool descriptors allows the standard LLVM interprocedural constant propagation pass to simplify some programs. In particular, any functions that are always passed the same global pool descriptor address will automatically have that parameter value (a link-time constant) substituted into their body. This makes the argument dead, allowing the standard LLVM dead argument elimination pass to remove it. For some programs composed primarily of global pools, this can reduce some of the pool descriptor arguments added to the program.

5.4.2 poolcreate/pooldestroy Placement

The algorithm described above places `poolcreate` and `pooldestroy` calls at the beginning and exits of each function. In practice, the lifetime of the data objects in a pool may begin at a later

point in the function and may end before the end of the function. Moving the pool create/destroy calls later and earlier within the function reduces the lifetime of objects in the pool[3]. We use simple depth-first traversals of the CFG to move the placement of create/destroy calls later and earlier within the *same* function where they were originally placed.

Starting from a `poolcreate`, we use a forward depth-first traversal of the CFG, searching for the first occurrence of a "real use" of a pool on any path, and place the `poolcreate` call at the immediate dominator of the CFG node containing that occurrence. Similarly, for `pooldestroy`, we use a backward traversal looking for the last occurrence of "real uses" and place the call in the immediate postdominator of each last occurrence on any path. A "real use" of a pool is a load, store, call, or poolalloc instruction that uses the pool descriptor, but *not* a `poolfree` instruction because any poolfree instructions that do not have any later "real uses" are essentially dead. Traversal is linear in the number of nodes and edges of the CFG.

```
int processlist(list* L) {
  list *A, *B, *tmp;
  Pool PD1, PD2;        // initialize pools
  poolcreate(&PD1); poolcreate(&PD2);
  splitclone(&PD1, &PD2, L, &A, &B);
  processPortion(A);    // Process first list
  processPortion(B);    // process second list

  while(A) { tmp=A->Next; poolfree(&PD1, A); A=tmp; }
  pooldestroy(&PD1);    // NOTE: this moved up

  while(B) { tmp=B->Next; poolfree(&PD2, B); B=tmp; }
  pooldestroy(&PD2);    // destroy pool PD2
}
```

Figure 5.7: After moving `pooldestroy(&PD1)` earlier

Figure 5.7 illustrates this placement, for the function processlist of our example. The call to `pooldestroy(&PD1)` has been moved earlier in the function, to immediately after the `while` loop that reads the Next field from nodes in PD1 pool. The `poolcreate` calls for both pools cannot be moved any later. Moving the `poolcreate` and `pooldestroy` calls interprocedurally [4] or into loops [26] can further reduce the lifetime of pools, but we have not yet implemented this.

[3]This refinement can also make it more likely that the optimization described in Section 6.1.2 can apply.

5.5 Experimental Results

Automatic Pool Allocation has two primary effects: first it enables new classes of macroscopic transformations (described in other chapters), second it has a direct performance effect on pointer intensive programs. Here we aim to evaluate the basic properties of the pool allocator and the code it produces. In Section 6.3, we evaluate the detail performance effect of the pool allocation transformation and show (by example) that the transformation succeeds in segregating data structures.

5.5.1 Methodology and Benchmarks

We implemented Automatic Pool Allocation as a link-time transformation using the LLVM Compiler Infrastructure (described in Chapter 2. All of the experiments in this section are compiled and optimized with the LLVM compiler, optionally run through the pool allocator, then converted to C code and compiled with GCC 3.4.2 for final code generation. All runtimes reported are the minimum user+system time from three identical executions of the program on an AMD Athlon MP 2100+ running Fedora Core 1 Linux at runlevel 3. Note that Chapter 6 describes a series of very simple transformations that can be used to boost program performance further.

For this work, we are most interested in heap intensive programs, particularly those that use recursive data structures. For this reason, we include numbers for the pointer-intensive SPECINT 2000 benchmarks, the Ptrdist suite [8], the Olden suite [109], and the FreeBench suite [115]. We also include a few standalone programs: Povray3.1 (a widely used open source ray tracer, available from `povray.org`), espresso, fpgrowth (a patent-protected, data mining algorithm [67]), llu-bench (a linked-list microbenchmark) [147], and "chomp" from the McGill benchmark suite. All but SPEC, fpgrowth and povray31 are available from `llvm.cs.uiuc.edu`.

Note that we elide many benchmarks from these suites that can not be effected by pool allocation, which occurs for for several reasons. Some of the benchmarks, including 181.mcf, 186.crafty, 256.bzip2, and several FreeBench benchmarks, have very few dynamic memory allocations. A few (e.g. 197.parser, 254.gap, 255.vortex) have custom memory allocators, which prevents disambigution of allocated memory objects and causes all objects to be placed in a single pool. As an experiment, we removed the custom memory allocator from 197.parser and replaced it with wrap-

pers that just call `malloc/free`; this is called 197.parser-b below. We can do this to 197.parser (but not the others) because its custom allocator has semantics identical to `malloc/free`. Finally, almost all the codes in the McGill benchmark suite have run times that are too small to be measured reliably.

5.5.2 Pool Allocation Statistics

Table 5.1 shows several basic statistics about pool allocation for each program. The StatPools column shows the number of static pools created in the program (when using Selective PA). The NumTH column shows the static number of type homogenous pools, and TH% is percentage of static pools that are type-homogenous. The DynPools column lists the number of dynamic pools created by the program. Tot Args and Max Args are the total number of formal arguments added to the program across all functions, and the maximum number for a single function.

Program	LOC	Stat Pools	Num TH	TH%	Dyn Pools	Tot Args	Max Args
164.gzip	8616	4	4	100%	44	1	1
175.vpr	17728	107	91	85%	44	23	4
197.parser-b	11204	49	48	98%	6674	76	16
252.eon	35819	124	123	99%	66	549	41
300.twolf	20461	94	88	94%	227	1	1
anagram	650	4	3	75%	4	0	0
bc	7297	24	22	32%	19	6	2
ft	1803	3	3	100%	4	0	0
ks	782	3	3	100%	3	0	0
yacr2	3982	20	20	100%	83	0	0
analyzer	923	5	5	100%	8	0	0
neural	785	5	5	100%	93	0	0
pcompress2	903	5	5	100%	8	0	0
llu-bench	191	1	1	100%	2	0	0
chomp	424	4	4	100%	7	10	8
fpgrowth	634	6	6	100%	3.4M	10	6
espresso	14959	160	160	100%	100K	191	13
povray31	108273	46	5	11%	14	290	4
bh	2090	1	0	0%	1	0	0
bisort	350	1	1	100%	1	1	1
em3d	682	12	12	100%	12	3	2
health	508	2	2	100%	2	4	2
mst	432	4	4	100%	4	0	0
perimeter	484	1	1	100%	1	1	1
power	622	3	3	100%	3	9	7
treeadd	245	6	6	100%	6	1	1
tsp	579	1	1	100%	1	1	1

Table 5.1: Basic Pool Allocation Statistics

The programs vary greatly in terms of the ratio of dynamic pool instances (Dyn Pools) to static

pools (Stat Pools). `fpgrowth` has a particularly high ratio because it creates a new pool (for a local search tree) in each call to a recursive function. The number of arguments added to the programs is generally modest. 252.eon has a large number of arguments added because the standard C++ library is statically linked in, providing a large amount of cold code.

The Th% column also shows that for most pools, DSA is able to successfully prove that memory in the pool is used in a type-consistent manner, which we have found true across a wide range of C programs. This allows intelligent alignment decisions, gives the pool runtime information about expected size for single objects, and enables other novel compiler techniques described in Chapters 6 and 7.

5.5.3 Pool Allocation Compile Time

Table 5.2 shows the compile times for pool allocation on programs bigger than 1000 lines of code. It breaks down this time into three components: the total time for DSA (which can be used by other clients as well), the time to compute the EBU graphs described in Section 5.2.3 (which are specific to pool allocation), and the time to perform the pool allocation transformation itself. The GCC column lists the time to compile the program with GCC 3.4.2 at -O3.

The total compilation time for pool allocation is extremely modest, taking less than 1.25 seconds in all cases on our Athlon 2100+. The largest amount of time is spent analyzing 252.eon (which has a large portion of the standard C++ library statically linked into it), followed by povray31; these are the only programs that took more than 1 second. Furthermore, much of the time is spent in DSA, which can be used for a variety of applications besides pool allocation. Our implementation of the EBU and PA passes have not been optimized substantially, so they could probably be further reduced. Overall, these compilation times are extremely small for a sophisticated interprocedural optimization.

To put these times in perspective, the GCC% column (computed as (Total/GCC)*100), shows that the pool allocation transformation takes 3% or less of the time taken by GCC to compile these programs. This is significant because GCC -O3 performs no cross-module optimizations and inlining is the only interprocedural optimization it performs within a module (thus it is very conservative compared to other interprocedurally optimizing compilers). Overall, we believe these

Program	LOC	GCC	DSA	EBU	PA	Total	GCC%
164.gzip	8616	2.67	0.02	0.01	0.01	0.03	1.1%
175.vpr	17728	9.39	0.06	0.03	0.05	0.14	1.5%
197.parser-b	11204	9.03	0.08	0.05	0.05	0.18	1.9%
252.eon	35819	131.13	0.51	0.30	0.42	1.23	0.9%
300.twolf	20459	17.21	0.09	0.07	0.03	0.19	1.1%
bc	7297	3.55	0.03	0.02	0.01	0.06	1.7%
ft	1803	0.68	0.01	0.01	0.01	0.02	2.9%
yacr2	3982	1.79	0.02	0.01	0.01	0.03	1.7%
espresso	14959	10.28	0.14	0.08	0.08	0.30	2.9%
povray31	108273	39.20	0.58	0.33	0.27	1.18	3.0%
bh	2090	0.85	0.01	0.01	0.01	0.01	1.2%

Table 5.2: Compile time (seconds) for programs > 1000 LOC

compilation times are quite acceptable for a production compiler.

Note that the effect of pool allocation on program performance and cache behavior is studied in detail in Section 6.3.

5.6 Related Work

The primary goal of the pool allocation transformation is to give the compiler some control over the layout of data structures in the heap. We achieve this using a context-sensitive points-to graph to distinguish data structure instances and object lifetimes. We first contrast this work with previous approaches for influencing the layout of heap objects, and then with previous work on partitioning the heap for automatic (region-based) memory management.

Chilimbi et al. [29] describe a semi-automatic tool called `ccmorph` that reorganizes the layout of homogeneous trees at runtime to improve locality. It relies on programmer annotations to identify the root of a tree and to indicate the reorganization is safe. We automatically identify and segregate instances of many kinds of logical data structures, but do not yet identify when a *runtime* reorganization would be safe. They also describe another tool, `ccmalloc`, which is a `malloc` replacement that accepts hints to allocate one object near another object. These hints only provides local information for an object pair and not any global information about entire data structures.

Hirzel et al. [74] describe a technique to improve the effectiveness of Garbage Collection by partitioning heap objects according to their connectivity properties. Unlike our work, their partitions

are not segregated on the runtime heap, are not directly related to distinct data structures, and the graph of partitions is restricted to be a DAG, which prevents fine grained partitioning of mutually recursive structures (like graphs).

Several proposed techniques aim to improve storage allocation or GC performance by relating objects based on their predicted lifetimes [68, 45, 12, 39, 30, 119]. These techniques use heuristics such as allocation site, call stack, or object size, combined with profiling information, to predict lifetime properties approximately. In contrast, our approach uses a more rigorous analysis to group objects both by structural relationships and statically derived lifetimes.

Other authors have developed techniques (usually profile-based) to reorganize fields within a single structure or place objects near each other to improve locality of reference [64, 20, 119, 28, 75]. These placement decisions are orthogonal to the choices made by Automatic Pool Allocation, and could therefore be combined with our transformation. This an important direction for future work.

There has been significant work on runtime libraries for region-based memory management [13], and on language mechanisms for manual region-based memory management as an alternative to garbage collection, e.g., Real-time Java [16], RC [58], Cyclone [78, 63], and others [58, 44, 17]. Compared with our approach, these library- or language-based techniques are much easier to implement, but require significant manual effort to use. In addition, although the region-based libraries and languages expose the relationship between objects and regions to the compiler, they do not expose any notion of higher-level data structures or how they relate to objects and regions. Therefore, the compiler does not obtain information about data structures and traversals that could enable optimizations on logical data structures.

There is a rich body of work on *automatic* region inference as a technique for memory management, for both functional [134, 133, 4, 66] and object-oriented languages [31, 26]. Unlike this body of our work, our primary goal is to segregate and control the layout of data structures in the heap for better performance and to enable subsequent compiler techniques that exploit knowledge of these layouts. We describe several optimizations (Chapters 6 and 7) that exploit data structure pools, and explore the performance implications of data structure segregation on program performance in some detail (Section 6.3). There are also some key technical differences between this prior work and ours. First, all these previous techniques except the work of Cherem and Rugina [26] are based on

type inference with a region-based type system. It does not appear straightforward to extend the type inference approaches to work for weakly-typed languages like C and C++, which can contain pointer casts, varargs functions, unions, etc., on which type information is difficult to propagate statically. In contrast, both our underlying pointer analysis and our transformation algorithm correctly handle all the complex features of C and C++, by distinguishing objects with known and unknown type (in the points-to graph) and by using a conservative and very efficient graph merging technique (the same as in DSA) to deal with potentially type-unsafe uses of pointers during the transformation. Second, using a pointer analysis as the basis for our transformation enables additional optimizations by exploiting the explicit relationship between a points-to graph and pools. Finally, the use of type inference and a rich type-system is not well suited for modern optimizing compilers, which are usually based on a mid-level or low-level internal representation supporting multiple source languages. Our approach is specifically designed for use in such compilers, and relies only a simple, mid-level intermediate representation and pointer analysis.

The work of Cherem and Rugina [26] was performed concurrently with ours and our approaches are technically similar in some key ways. They describe a region inference approach for Java based on a flow-insensitive, context-sensitive points-to analysis. Because their primary focus is automatic memory management, they are much more aggressive about computing region lifetimes, including loop-carried regions. Our regions *can* be placed as flexibly as theirs, but we use a simpler placement analysis. Like the type-inference approaches, however, their work also does not support weakly typed languages like C. Although the underlying pointer analysis could be extended to do so (using our approach, for example), we believe the transformation would be more difficult to extend. Furthermore, they too focus on automatic memory management, and do not explore the impact of their work on memory hierarchy performance or consider other optimizations that could exploit their region information. We expect that our optimization techniques could be fruitfully combined with their region inference algorithm for Java programs.

There is a wide range of work on techniques for stack allocation of heap objects as well as techniques for static garbage collection, both of which are based on analyzing the lifetimes of objects in programs (e.g., see [14, 122, 79] and the references therein). These techniques do not attempt to analyze or control the layout of logical data structures in the heap *per se*, and are largely

orthogonal to our work. A minor exception is that our optimization to eliminate `poolfree` for a pool (when there are no intervening allocations before the subsequent `pooldestroy`) essentially replaces explicit deallocation with static reclamation of memory in the pool. This is the inverse of (and much more limited than) the work on static GC, which aims to replace or optimize runtime GC.

5.7 Research Contributions of Automatic Pool Allocation

The primary contribution of Automatic Pool Allocation is a practical, efficient compiler algorithm to segregate distinct instances of logical data structures into separate pools in the heap. Our algorithm and implementation make the following specific contributions:

(i) We show that the algorithm succeeds in segregating recursive data structures on the heap, providing a substantial performance improvement for several programs. We experimentally find that the algorithm improves the performance of several programs by 10-20%, speeds up two by about 2x and two others by about 10x, and explain the source of these improvements.

(ii) Unlike previous approaches related approaches [134, 133, 4, 66, 31, 26], *all* of which require a type-safe input program, Automatic Pool Allocation supports the full generality of C and C++ programs (including indirect function calls, mutually recursive functions, variable argument functions, lack of

(iii) Our algorithm is the first to perform region inference based on a scalable pointer analysis (DSA), which allows us to partition heap data by *reachability*. Work concurrent to ours [26] uses a somewhat similar approach, but uses a non-scalable analysis, does not handle global variables at all, requires type-safety, and has other limitations compared to our approach.

(iv) We present a simple strategy for correctly handling indirect function calls in arbitrary C programs without making the core transformation more complex.

(v) The algorithm computes static mapping information from pointers to the pools that they point into. We are the first to demonstrate that region inference and this mapping information can be used to support aggressive follow-on techniques like Transparent Pointer Compression.

In addition to the research contributions, we show that the analysis and transformation required to perform this optimization both require very little compile time or memory. We feel that the amount of resources used is quite reasonable for aggressive optimizing compilers. Finally, Chapter 6 provides a detailed evaluation of the performance effect of Automatic Pool Allocation.

Chapter 6

Optimizing Pool Allocated Code

Chapter 5 describes the basic pool allocation transformation, along with the refinements which allow it to produce reasonably efficient code. While pool allocation itself can produce a strong performance improvement for many programs that heavily use heap data structures, Chapter 5 did not attempt to use any of the extra information provided by pool allocation to further improve the performance of the program, and did not evaluate the performance impact that pool allocation itself has.

This chapter is dedicated to simple techniques which can take a pool allocated program and improve its performance. These techniques generally make use of the partitioning of memory into pools to allow the optimizations to conclude the behavior of a subset of the heap memory in the program. While some of these techniques could in theory be applied to programs that are not pool allocated (e.g. the bump pointer optimization), doing so would only allow them to be applied in unrealistic cases (e.g., the program never deallocated any memory).

In addition to simple pool optimizations, this chapter also discusses pool collocation (assigning more than one DS Node to a pool), extensions of the pool allocation algorithm to support collocation, several example heuristics, and our experience with collocation.

Finally, we evaluate the performance impact of pool allocation with and without these optimizations, and evaluate the contribution of each one.

6.1 Pool Optimizations

We describe four optimizations that exploit the partitioning of heap objects by the pool allocator into pools, and the control we have over the pool runtime library. The benefits of all four optimizations are evaluated in Section 6.3.

6.1.1 Avoiding Pool Allocation for Singleton Objects: SelectivePA

The simplest optimization we proposte attempts to avoid pool allocating DS nodes that will have a single memory object allocated from it. Pool allocation suffers from two performance disadvantages when pool allocating these singleton pools: First, it adds overhead to the program by requiring the creation, destruction and potential argument passing of pool descriptors.

Second, and more importantly, the pool allocation runtime is optimized to handle collections of allocations, so it does not perform well in time or space for singleton allocations. In particular, on the first allocation, it allocates a large chunk of memory sufficient to hold the requested memory plus several more allocations. If there is is only ever one dynamic allocation from a pool, this extra memory allocated is wasted as long as the pool is live.

We identify potentially singleton pools with a simple heuristic: we classify all **H** nodes that are not pointed to by any other memory object (including itself) as singleton objects. This conservative approximation perserves pool allocation for common cases (such as recursive structures, stack or global arrays that point to a large number of nodes, etc) while filtering out some obviously bad cases.

In practice, we find that this triggers the most for functions that allocate a dynamic array of memory on entry to the function and deallocate the memory before exit. If the array had a statically bound size that was known to be small, it would be reasonable to stack allocate the object. Because these objects have an unknown (and potentially very large) size, we just preserve the malloc/free.

We name this optimization “Selective PA”.

6.1.2 `poolfree` Elimination: PoolFreeElim

The `pooldestroy` operation atomically releases all memory from a pool, regardless of whether or not the program has explicitly freed it. As mentioned in Section 5.2.2, this can actually fix memory leaks in some cases by reclaiming memory that would be otherwise leaked. The `poolfree` elimination optimization is based on the observation that we can actually *induce* memory leaks into the program if we can prove that they will not increase the peak heap size of the program.

This optimization can be applied in many cases. Intuitively, many short-lived data structures have a "build-use-destroy" pattern, in which all allocations happen before any deallocations. In these cases (e.g. the example in Figure 5.2), if no memory is allocated from any pool between the `poolfree` and the pool destroy for the pool, the `poolfree` operations can be eliminated. For the example in Figure 5.2, this leaves us with the code shown in Figure 6.1.

```
int processlist(list* L) {
  list *A, *B, *tmp;
  Pool PD1, PD2;
  poolcreate(&PD1);
  poolcreate(&PD2);
  splitclone(&PD1, &PD2, L, &A, &B);
  processPortion(A);  // Process first list
  processPortion(B);  // process second list

  while (A) { tmp = A->Next; A = tmp; }
  while (B) { tmp = B->Next; B = tmp; }

  pooldestroy(&PD1);  // destroy pool (including list nodes)
  pooldestroy(&PD2);  // destroy pool (including list nodes)
}
```

Figure 6.1: Figure 5.2 after eliminating `poolfree` calls

Note that elimination of the `poolfree` operations in the deallocation loops actually makes the loops *output free*. Because of this, standard algorithms for aggressive dead code elimination (e.g. LLVM's, `-adce` pass) can be used to eliminate the dead loops, leaving us with the code in Figure 6.2.

The `poolfree` elimination optimization is beneficial for two reasons: first, it removes some unneccesary manipulations of the freelist (which is minor), second, this optimization occasionally allows the removal of entire *traversals* of data structures, such as in the case above.

In order to detect unneccesary `poolfree` calls, we perform a simple intraprocedural backward dataflow analysis (per pool) from the pool destroy calls for the pool, identifying basic blocks in

```
int processlist(list* L) {
  list *A, *B, *tmp;
  Pool PD1, PD2;
  poolcreate(&PD1);
  poolcreate(&PD2);
  splitclone(&PD1, &PD2, L, &A, &B);
  processPortion(A);  // Process first list
  processPortion(B);  // process second list
  pooldestroy(&PD1);  // destroy pool (including list nodes)
  pooldestroy(&PD2);  // destroy pool (including list nodes)
}
```

Figure 6.2: After eliminating `poolfree` calls and dead loops

the CFG which have no `poolalloc` calls on any path from their block to a pooldestroy. We then remove any `poolfree` calls to the current pool in these blocks.

Note that the refinement describe in Section 5.4.2 can positively interact with `poolfree` elimination, by moving the pooldestroy for a pool above calls to `poolalloc` from other pools. Also note that this could be enhanced to use interprocedural analysis to increase the number of opportunities to apply the optimization, and could use more aggressive interprocedural techniques to remove calls in (for example) recursive functions that are used to delete the nodes in a data structure.

We name this optimization "PoolFreeElim".

6.1.3 Avoid Object Header Overhead: Bump-Pointer

The pool allocator runtime supports the full set of heap operations, including malloc/free/realloc etc. This support is required to handle the full generality of programs, but not all pools need this generality. In particular, one of the costs of this generality is that objects allocated from the pool need to include a header word, which indicates the size of the object and includes bookkeeping information to be used when freeing objects. In particular, consider a pool containing 16 byte objects. The objects will be laid out in the pool as shown in Figure 6.3.

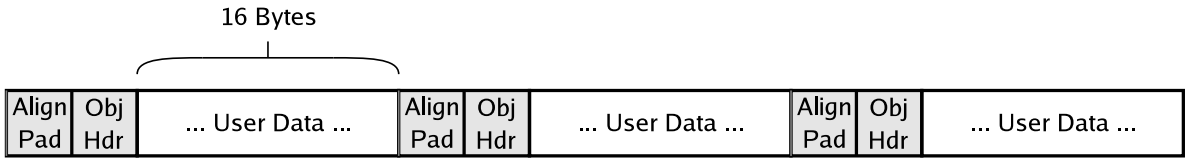


Figure 6.3: Standard Pool of 16-byte Objects with Default 8-Byte Alignment

This diagram shows the header word and alignment padding words required for book-keeping and required to keep the user data 8-byte aligned (which is the default alignment for standard malloc implementations and the pool allocator runtime). If the CPU has 32-byte cache lines, it can hold 1.5 objects per cache line.

The bump-pointer optimization applies to pools which are allocated from but can be proven to never have memory deallocated back to them. In this case, the header word is completely unneccesary: all of the bookkeeping for deallocation is unneeded for the pool. The bump pointer optimization transforms all calls to pool library functions to call bump-pointer versions instead (e.g., change `poolalloc` to `poolalloc_bp`, `poolcreate` to `poolcreate_bp`, etc). The bump-pointer interface to the pool library supports only a simple light-weight allocator with an extremely fast allocation path (no free-lists to search), and does not use object headers in objects. After the bump-pointer optimization, the pool in Figure 6.3 is laid out as shown in Figure 6.4.

Figure 6.4: Bump-Pointer Pool of 16-byte Objects with Default 8-Byte Alignment

The advantage in this case is extreme: instead of fitting 1.5 objects into each cache line, we are able to fit 2 objects into a cache line. Clearly the benefit of this optimization is larger with smaller objects, and smaller with larger ones. Finally, note that this optimization interacts with the PoolFreeElim optimization described in Section 6.1.2: if it is able to remove all of the `poolfree` calls for a pool, we will be able to convert the pool to use a bump pointer.

Our implementation of the bump-pointer optimization is currently very simple. First it identifies all poolcreate calls in the program, using them to identify the pool descriptors in the program. Second, it walks the def-use chains for the pool descriptors, inspecting every use of the descriptor. If the pool descriptor is never passed into a non-pool-allocator function or passed to a `poolfree` call, it is promoted to use the bump pointer pool interfaces. This very simple implementation could be extended in several obvious ways, none of which have been implemented yet.

6.1.4 Avoiding Alignment Padding: AlignOpt

As mentioned above, the pool allocator and most standard `malloc` implementations all return 8-byte aligned memory by default. The problem is that traditional heap libraries must return memory that can be used to hold any data-type supported by the processor. Most RISC machines require 8-byte data to be 8-byte aligned, and even processors that support unaligned data (e.g., the X86 line) generally do so at a significant performance penalty (e.g., the Alpha traps to the operating system to emulate it).

This alignment restriction requires that the allocator insert inter-object padding in cases where the available space for an object is not suitably aligned. For example, on a 32-bit machine, all allocations of $8n$ bytes generally need a 4-byte alignment pad (allocations of $8n + 4$ bytes do not require an alignment pad). The example in Figure 6.3 illustrates an example for 16-byte objects.

The AlignOpt optimization uses the DSNodes computed by DSA to infer when it is safe to reduce alignment from 8-bytes to 4 bytes for a pool. In particular, if a pool is type-homogenous, and if the type for the pool does not contain any data that requires 8-byte alignment, the `poolinit` (or `poolinit_bp`) call is modified to request 4-byte alignment instead of 8-byte alignment. For the example shown in Figure 6.3, this changes the memory layout to that shown in Figure 6.5.

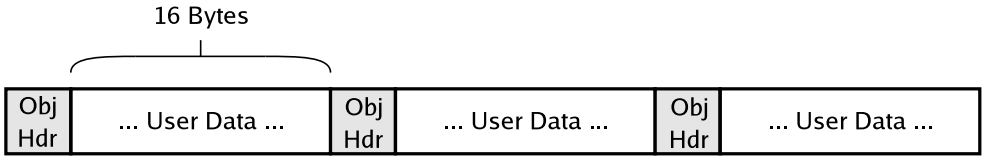


Figure 6.5: Normal Pool of 16-byte Objects with Reduced 4-Byte Alignment

Like the bump-pointer optimization, this optimization reduces inter-object padding to increase the number of objects that will fit into a cache line (in this case, from 1.5 to 1.75). Like the bump-pointer optimization, it has a more dramatic performance effect for smaller objects than large ones. Note that this optimization works together with the bump-pointer optimization. The alignment optimization applies in cases bump-pointer optimization doesn't (and visa-versa), and they can be applied together. If the bump-pointer optimization has been applied to a pool, the alignment optimization helps for pools that contain objects of size $8n + 4$ (e.g. 12 bytes) instead of $8n$.

6.1.5 Tail Padding Optimization

In C, the size of a structure is determined by the elements in the structure, padding to ensure each element in the structure is aligned to an appropriate boundary, and padding inserted at the end of an object to ensure that all elements in an array of the structure will be appropriately aligned. For example, Figure 6.6 contains a simple example that requires 4 bytes of tail padding on a 32-bit machine:

```
struct DoubleList {    // sizeof(struct DoubleList) == 16
   double Data;
   struct DoubleList *Next;
};
```

Figure 6.6: Example Structure with Tail Padding

Like other padding, tail padding for a structure wastes cache capacity with unneccesary data, reducing its effective size. We propose (but have not implemented) that the compiler detect type-homogenous pools that contain objects with tail padding and transform them to pass the tail padding amount into the `poolcreate` call. Given this, the `poolalloc` call can implicitly subtract the tail-padding amount from any allocation request. Even if the bump-pointer optimization and alignment optimizations fail, this can eliminate interobject padding by placing object headers in the tail padding (in this case, reducing the object size to 12-bytes eliminates the need for an alignment pad).

Currently our analysis does not keep track of whether or not `memcpy/memmove/memset` are used on memory allocated from a pool. Without this information, eliminating tail padding is not safe, as these can copy and clobber anything put into the tail pad.

6.2 Collocation of DS Nodes into Shared Pools

The basic pool allocation algorithm provides a general framework for segregating data structures in the heap but never collocates two DS nodes into the same pool. Intuitively, it seems that collocation can improve the performance of programs that often access nodes from two different data structures in an interlaced fashion. For example, if a program contains a linked list, where every list node contains a pointer to the data, a common traversal pattern may be to walk the list and dereference

the pointer at each node. collocating the list with the data may improve locality.

There are two aspects to implement this in the pool allocator framework: 1) allowing multiple DS Nodes to be associated with the same pool, and 2) heuristics to determine when to merge multiple DS Nodes into a single pool. We describe these in Sections 6.2.1 and 6.2.2 below. Section 6.2.3 describes our experiences with collocation.

6.2.1 Algorithm Extensions to Support Collocation

We can easily adapt the pool allocation algorithm to support collocation simply by changing line 10 of Figure 5.4 to only insert `poolcreate` and `pooldestroy` for one of the nodes being collocated, and initialize the "pdmap" entries for the other nodes to point to the common descriptor. Since heap objects are laid out separately and dynamically within each pool, collocating objects can give the compiler some control over the internal layout of data structures. Even more sophisticated control might be possible using additional techniques (e.g., [20]) on a per-pool basis.

6.2.2 Node Collocation Heuristics

In our implementation, we experimented with two static heuristics for collocating **H** node into a common pool. For these heuristics, we define a *Collection* to be either a node with $A = true$, or any non-trivial strongly connected component (SCC) in the DS Graph. A non-trivial SCC is one containing at least one cycle, including self-cycles. Given this, any **H** node reachable from a Collection represents a set of objects which may be visited by a single traversal over the objects of the collection.

The NoCollocation Heuristic

This is the default heuristic used which assigns each DSNode to its own pool, as described in Chapter 5.

The OnePoolPerCollection Heuristic

All candidate **H** nodes in a collection are assigned to a single pool. Any other **H** node reachable from a collection (without going through another collection) is assigned to the same pool as the

collection. This choice effectively partitions the heap so that each minimal "traversable" collection of objects becomes a separate pool. Intuitively, this gives the finest-grain partitioning of recursive data structures, which are often hierarchical. It favors traversals over a single collection within such a hierarchical (i.e., multi-collection) data structure.

The MaximalDS Heuristic

Assign a maximal connected subgraph of the DS graph in which all nodes are **H** nodes to a single pool. This partition could be useful as a default choice if there is no information about traversal orders within and across collections. In particular, this can help if a program creates a complex connected data structure consisting of multiple DS nodes and traverses it in the same order as it was created. This heuristic puts all nodes in a single pool, allowing such a traversal to be linear in memory (if objects are laid out by the library in allocation order).

6.2.3 Experiences with Node Collocation

Our implementation of pool allocation supports flexible and pluggable collocation policies, and we experimented with the options above and several other (ad-hoc) choices. In practice, however, we found that using static collocation heuristics rarely outperform (and often do much worse than) assigning each **H** node to a separate pool. In our experiments we find several explanations which contribute to this effect:

- Collocation interferes and often disables the pool optimizations described in Section 6.1. In particular, merging pools often breaks type homogeneity, can lose the "never freed to" property, etc.
- Static heuristics may not be enough. In particular it is possible that profile information could be used to tune the collocation choices to the program being optimized. We have not experimented with using profile information to drive collocation decisions.
- Collocation can, but does not necessarily, improve the amortized locality for programs that uses data structure *traversals* (in contrast to occasionally accessing single associated memory objects), even when achieving "perfect" collocation.

The third observation is the most important, so we expand on it here. In particular, consider a linked list of pointers to doubles. The memory layout of this data structure without collocation might be as shown in Figure 6.7 (ignoring inter-object padding and object headers).

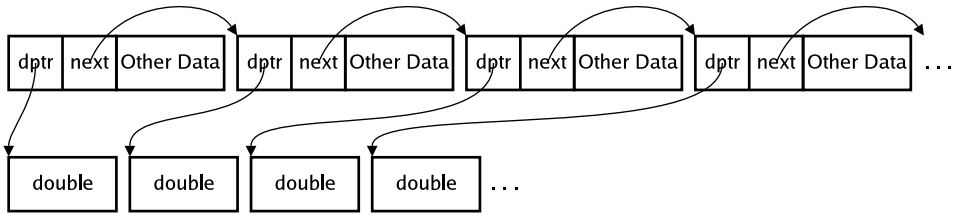


Figure 6.7: Linked List of Doubles without Node Collocation

After collocation of these two pools, assuming the best case node interleaving occurs, the pool would be laid out as shown in Figure 6.8.

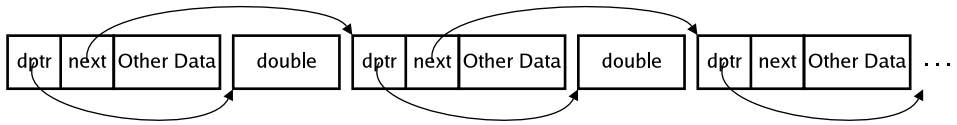


Figure 6.8: Linked List of Doubles with Perfect Node Collocation

If the program frequently traverses the linked list, there are two possible traversal strategies it could use. First, even if the program does not use the double nodes during a traversal of the list, they are pulled into the cache when the list nodes are accessed. If this is the most important traversal, collocation of this data structure would clearly be detrimental to effective cache capacity for this list. This effect can potentially be avoided through the use of profile information and/or smart heuristics. The second possible traversal *does* access the double every time the list node is accessed.

In this traversal, node collocation intuitively should help locality by avoiding a potential cache miss accessing the double. Unfortunately, two issues make this significantly less likely to occur than we would like. First, this behavior only holds in the "perfect" case above, where we succeed at putting the dereferenced node immediately after the single node that points to it. In practice, however, it is likely that the program either mutates the "dptr" pointers during the lifetime of the list, or it has multiple list nodes that point to the double nodes (otherwise the double should have been inlined into the list).

The second issue is that, in many cases when the perfect situation occurs for collocation, collocation will not actually reduce the amortized number of cache lines accessed by the program. In particular, assume the list is a 16-byte object, the double is an 8-byte object, a cache line is 32-bytes, and assume there is no inter-object padding for these structures. In this case, before collocation, two list nodes or four doubles fit on a single cache line. After collocation, 1.5 line nodes and one double fit on a cache line. If the program performs a traversal where it gets inter-node reuse from the cache, both organizations of memory will cause the same number of cache lines to be read from memory.

Though collocation does not help the case above, we can easily come up with other cases where it would help, assuming perfect colocation. In particular, if a data structure is not traversed in perfect memory order (e.g. querying a binary search tree), collocation could significantly reduce the number of cache misses for data pointed to by the tree nodes.

Clearly there are many variables that interact and may effect the profitability of collocation. Though our experiments have not shown an advantage to using collocation, more aggressive techniques (e.g. using profile information) that avoid bad cases could show substantial locality benefits. We leave full investigation of collocation algorithms and benefits to future work.

6.3 Pool Allocation and Optimization Performance Results

The pool allocation optimizations described in Section 6.1 are directly aimed at improving the performance of the application. As such, we are interested in several aspects: 1) How often do the optimizations trigger? 2) What is the aggregate performance impact of the optimizations? 3) What contribution to the aggregate impact does each optimization make? Also, because Chapter 5 did not evaluate the performance impact or overhead of pool allocation, we do so here.

To quantify these aspects of the optimizations, we applied and ran the optimizations on the same set of programs and on the same machine as was used to evaluate the pool allocator in Section 5.5.

6.3.1 Implementation and Evaluation Framework

We implemented Automatic Pool Allocation as a link-time transformation using the LLVM Compiler Infrastructure (Chapter 2). Our system compiles source programs into the LLVM representation (for C and C++, we use a modified version of the GCC front-end), applies standard intraprocedural optimizations to each module, links the LLVM object files into a single LLVM module, and then applies interprocedural optimizations. At this stage, we first compute the complete Bottom-up DS graphs and then apply the Pool Allocation algorithm. Finally, we run a few passes to clean up the resulting code, the most important of which are interprocedural constant propagation (IPCP), to propagate null or global pool descriptors when these are passed as function arguments, and dead argument elimination (to remove pool pointer arguments made dead by IPCP). The resulting code is compiled to either native or C code using one of the LLVM back-ends, and linked with any native code libraries (i.e., those not available in LLVM form) for execution.

6.3.2 Number of Pool Optimization Opportunities

Program	BP	BP%	PFE
164.gzip	1	25%	9
175.vpr	27	25%	29
197.parser-b	3	6%	0
252.eon	0	0%	28
300.twolf	61	65%	1
anagram	2	50%	0
bc	3	13%	0
ft	2	67%	0
ks	3	100%	0
yacr2	7	35%	0
analyzer	5	100%	0
neural	5	100%	0
pcompress2	0	0%	0

Program	BP	BP%	PFE
llu-bench	1	100%	0
chomp	0	0%	0
fpgrowth	0	0%	0
espresso	1	1%	3
povray31	6	13%	28
bh	1	100%	0
bisort	1	100%	0
em3d	6	50%	0
health	2	100%	0
mst	4	100%	0
perimeter	1	100%	0
power	3	100%	0
treeadd	2	33%	0
tsp	1	100%	0

Figure 6.9: Statistics for Pool Optimizations

Table 6.9 shows the static number of pools that can use a bump pointer after poolfree elimination (BP), and number of `poolfree` calls deleted when PoolFree Elim is enabled (PFE). The table shows that in many programs (the larger such examples are vpr, twolf, yacr2, and povray), a significant fraction of pools are identified as eligible bump-pointer pools (no frees occur to a pool), even with our simple detection algorithm. For vpr, twolf and povray, this is enabled by the elimination of several `poolfree` operations. This elimination indicates the presence of the build-use-destroy

pattern explained in Section 6.1.2. In 175.vpr, for example, pool allocation eliminates 29 poolfree calls. Overall, the table shows that though these optimizations are very simple, they do trigger a large number of times, building off of the segregation of memory performed by the pool allocator.

Program	GCC	NoPA	One - Pool	OnePool Run %	Only - OH	OnlyOH Run %
164.gzip	25.11	28.16	28.44	101.0%	28.17	100.0%
175.vpr	10.54	10.88	10.86	99.8%	10.87	99.9%
197.parser-b	12.59	12.42	17.86	142.7%	13.36	106.7%
252.eon	1.15	0.86	0.85	98.8%	0.88	102.3%
300.twolf	20.26	20.10	19.98	99.4%	20.50	102.0%
anagram	3.46	3.02	3.01	99.7%	3.02	100.0%
bc	1.71	1.55	1.48	95.5%	1.71	110.3%
ft	63.74	68.73	66.08	96.1%	68.94	100.3%
ks	4.56	4.43	5.30	119.6%	4.39	99.1%
yacr2	3.76	3.86	3.94	102.0%	3.89	100.8%
analyzer	324.54	312.25	314.69	100.8%	313.69	100.5%
neural	88.82	87.34	87.35	100.0%	87.60	100.3%
pcompress2	38.61	37.77	37.44	99.1%	38.04	100.7%
llu-bench	106.63	106.50	108.86	102.2%	106.76	100.2%
chomp	17.26	16.71	10.63	63.6%	16.82	100.6%
fpgrowth	36.27	36.62	36.49	99.7%	39.30	107.3%
espresso	1.25	1.22	1.20	98.3%	1.26	103.3%
povray31	9.41	9.79	9.69	98.9%	9.81	100.2%
bh	14.02	9.33	9.32	99.9%	9.35	100.2%
bisort	12.59	13.06	13.14	100.6%	13.20	101.1%
em3d	9.55	6.80	6.76	99.4%	6.80	100.0%
health	14.11	13.99	13.39	95.7%	13.98	99.9%
mst	12.79	13.14	13.23	100.7%	13.34	101.5%
perimeter	3.02	2.92	2.58	88.4%	3.00	102.7%
power	4.61	2.91	2.93	100.7%	2.92	100.3%
treeadd	17.48	17.41	17.29	99.3%	17.6	101.1%
tsp	7.17	7.24	7.08	97.8%	7.42	102.5%

Table 6.1: Baseline (NoPA), allocator, and overhead comparisons

6.3.3 Performance Baseline, Allocator Influence, and Overhead

Table 6.1 shows data to characterize the baseline we use for comparison and isolate the overheads added to a program by pool allocation. The GCC column is the execution time of the program with the GCC 3.4.2 compiler (at -O3). The NoPA column is the program compiled with LLVM using exactly the same sequence of transformation and cleanup passes as we do for pool allocation (described in Section 6.3.1), but with the pool allocator and all pool-based optimizations disabled. Using NoPA as a baseline for comparison below isolates the speedup of the pool allocator transformation and its optimizations by factoring out the impact of other LLVM compiler passes. Comparing GCC to NoPA shows that the LLVM-generated code is no worse than 12% slower than GCC code and is sometimes much better. This indicates that the code quality of NoPA is reasonable to use a baseline for comparisons.

Another key question is how the difference between the allocator in our pool runtime library (used after pool allocation) and the standard libc malloc library (used by NoPA) affect the comparisons. This is significant because our pool library implementation is currently not thread-safe (though it is otherwise fully general), and this or other implementation details could skew the results in our favor. To measure this, we transformed the programs to allocate out of a single global pool (this transformation does not add pool arguments or other overhead to the program), effectively using our allocator to replace malloc and free for the program (the OnePool column). Comparing with NoPA shows that in all but 4 cases (197.parser-b, ks, chomp and perimter), OnePool is within about 5% of NoPA. The large slowdown for parser-b occurs because we use a singly-linked free list and the order of frees prevents coalescing adjacent free blocks. `chomp` is much faster with our allocator because our allocator has a fast path for fixed size allocations (to exploit type homogeneous pools) and nearly all allocations in `chomp` are (multiples of) this fixed size. As shown below, in all cases except perimeter, any such advantages from our runtime library (even chomp) are much smaller than the aggregate performance improvements due to pool allocation.

Finally, the OnlyOH column aims to isolate the performance overheads in the transformed code, namely, extra pool arguments on functions and initializing and destroying pool descriptors. It is computed by pool-allocating the program, but modifying the runtime library so that poolalloc/free simply call malloc/free. Comparing to NoPA shows that this overhead is negligible or quite low (less than about 5%) in nearly all cases, but is slightly higher in 197.parser-b (7%), bc (10%), and fpgrowth (7%). The pool allocator must overcome this overhead to provide a net performance improvement.

6.3.4 Aggregate Performance Effect of Pool Allocation & Optimizations

Figure 6.10 and Table 6.2 shows the program running time and speedups (relative to NoPA) for automatic pool allocation alone (BasePA) and for pool allocation with all pool-based optimizations (FullPA). FullPA therefore represents the aggregate performance impact of this work. As the table shows, FullPA improves the performance of many programs from 5% to 20%, improves analyzer and llu-bench by roughly 2x, and ft and chomp more than 10x (see Section 6.3.7 for an analysis of ft and chomp). In no case does FullPA hurt the performance of other programs relative to NoPA.

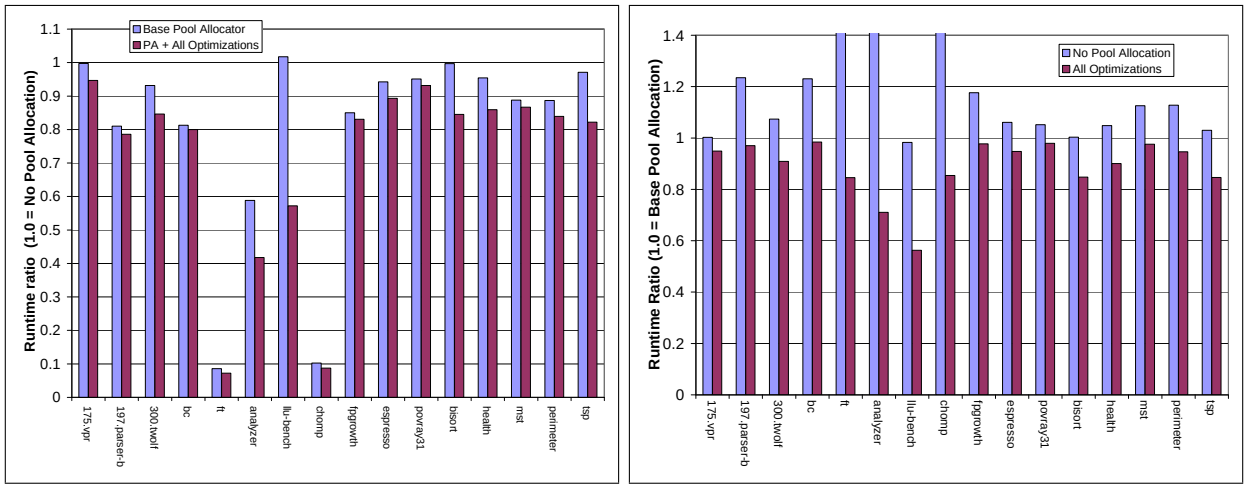


Figure 6.10: Aggregate execution time ratios (Left 1.0 = NoPA, Right 1.0 = BasePA)

Not surprisingly, there is no obvious correlation between the speedups obtained and the number of static or dynamic pools.

The causes and breakdown of these improvements are studied below. The charts in Figure 6.10 are shown with two different baselines. The chart on the left makes it easy to see the net impact of pool allocation and the pool optimizations, and the chart on the right allows inspection of the performance effect of the pool optimizations over and above what pool allocation itself does. This shows that the pool optimizations contribute significant performance improvements to these programs.

6.3.5 Performance Contribution of Individual Pool Optimizations

Figures 6.11 and 6.12 shows the runtime ratio of each program with one optimization disabled at a time, and compares it to two baselines (NoPA for the former and FullPA for the later). This shows how much the program slows down when a particular optimization is disabled, which is correlated to how much the optimization helps the performance of the code. Note that if two optimizations can provide the speedup (e.g. either use of alignment-opt or bump-pointer to reduce inter-object padding), disabling either will not show a slowdown. Despite this, this analysis does provide useful insight into the effect of the optimizations.

All of the optimizations except SelectivePA contribute noticeable improvements to at least one program. SelectivePA provides no significant speedup but does not hurt performance and it is useful

Program	NoPA	BasePA	BasePA/ NoPA	FullPA	FullPA/ NoPA
164.gzip	28.09	27.93	0.99	28.40	1.01
175.vpr	10.88	10.85	1.00	10.30	0.94
197.parser-b	12.52	10.14	0.81	9.84	0.79
252.eon	0.86	0.84	0.98	0.84	0.98
300.twolf	20.10	17.59	0.88	17.01	0.85
anagram	3.02	3.00	0.99	3.00	0.99
bc	1.55	1.26	0.81	1.24	0.80
ft	68.73	5.89	0.09	4.98	0.07
ks	4.43	4.38	0.99	4.39	0.99
yacr2	3.89	3.89	1.01	3.87	1.00
analyzer	312.25	183.64	0.59	130.53	0.42
neural	87.60	87.33	1.00	87.15	1.00
pcompress2	38.04	37.52	0.99	37.68	1.00
llu-bench	106.50	108.37	1.02	60.96	0.57
chomp	16.71	1.71	0.10	1.46	0.09
fpgrowth	36.62	31.13	0.85	30.42	0.83
espresso	1.22	1.15	0.94	1.09	0.89
povray31	9.79	9.31	0.95	9.12	0.93
bh	9.33	9.41	1.01	8.88	0.95
bisort	13.06	13.02	1.00	11.04	0.85
em3d	6.80	6.82	1.00	6.62	0.97
health	13.99	13.35	0.95	12.02	0.86
mst	13.14	11.67	0.89	11.39	0.87
perimeter	2.92	2.59	0.89	2.45	0.84
power	2.91	2.91	1.00	2.91	1.00
treeadd	17.41	17.19	0.99	16.85	0.97
tsp	7.24	7.03	0.97	5.95	0.82

Table 6.2: Run time (seconds) and runtime ratios vs. NoPA

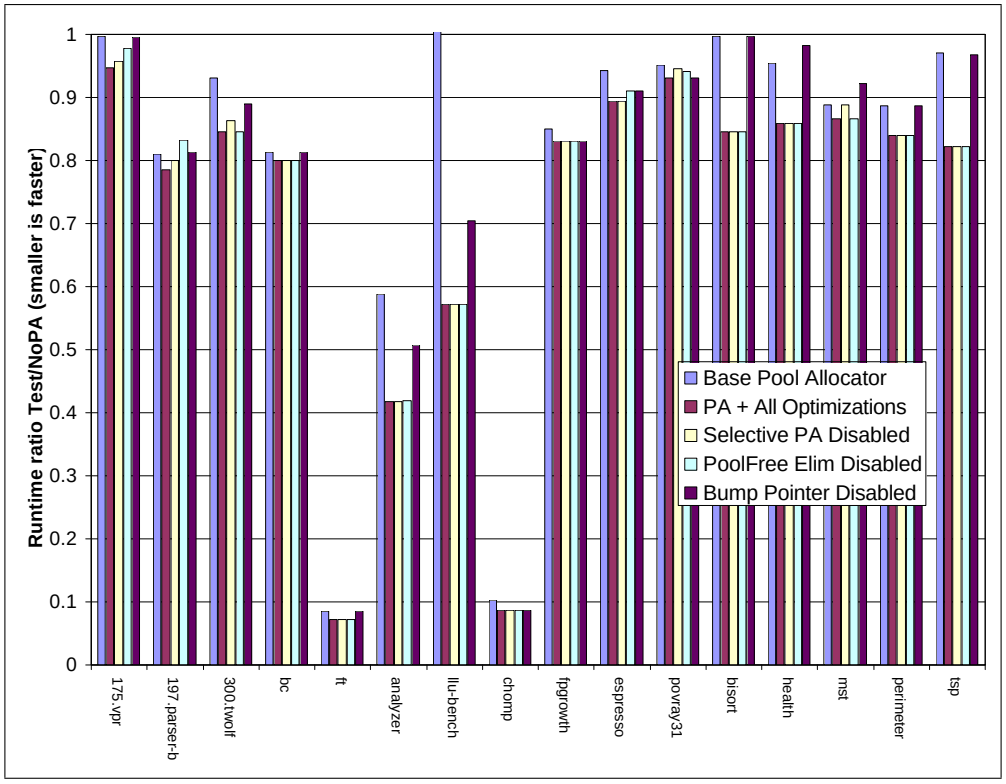


Figure 6.11: Pool Optimization Contributions (1.0 = No Pool Allocation)

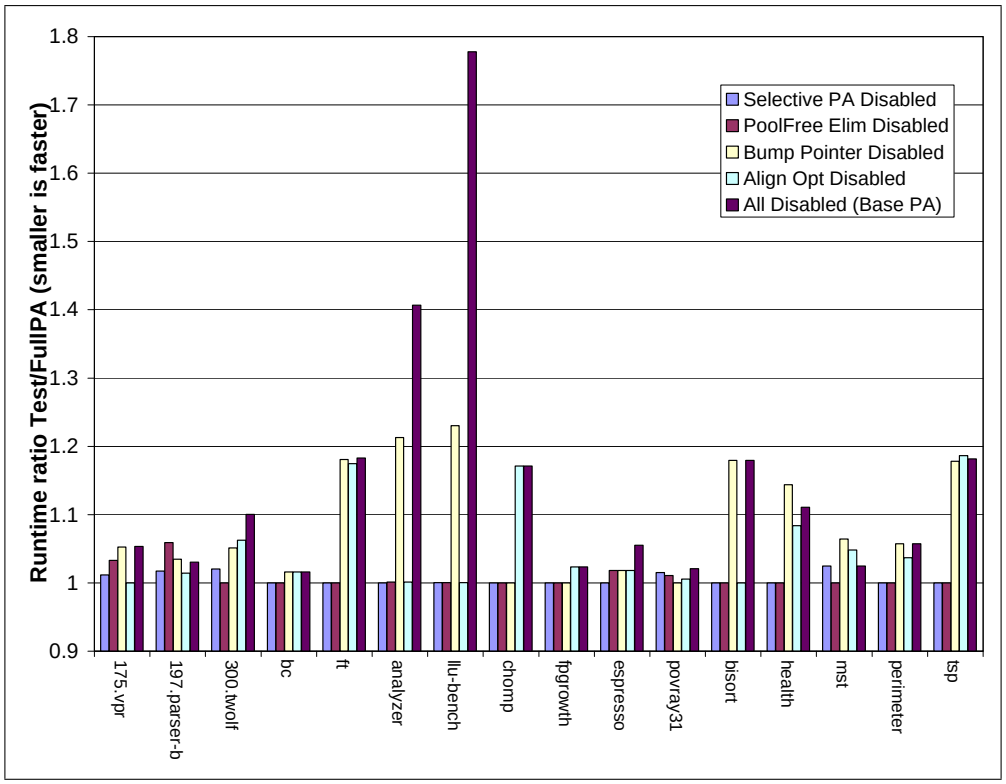


Figure 6.12: Pool Optimization Contributions (1.0 = PA with all PoolOpts)

because it can improve memory consumption significantly in some cases. The poolfree optimization improves 175.vpr, 197.parser-b, espresso, and povray31. The bump pointer optimization appears to be the most significant of the three, being particularly valuable to 175.vpr, 300.twolf, ft, analyzer, llu-bench, and several Olden programs. Close inspection of 175.vpr is particularly interesting: BasePA is not faster than NoPA, but a combination of poolfree elimination and the bump pointer optimization reduces the runtime of the program to 95.7% of NoPA (SelectivePA reduces it further to 94.6%). Finally, several programs benefited from the alignment optimization, particularly ft, chomp, health and tsp.

The speedup potential of these simple pool optimizations are particularly notable because they are all very simple optimizations, but can only be performed only once the heap has been segregated into pools.

6.3.6 Cache and TLB Impact of Pool Allocation

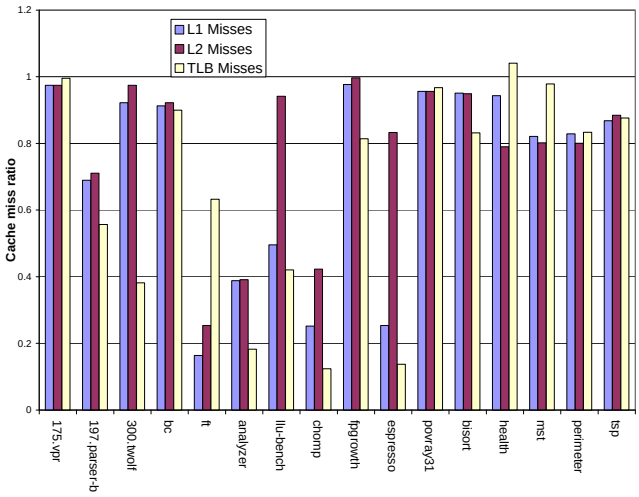


Figure 6.13: L1/L2/TLB Cache Miss Ratios

Figure 6.13 shows the measured cache miss ratio of FullPA compared to NoPA. The figure includes measurements showing the number of accesses that miss the Athlon's L1 D-cache, the number of accesses that miss the L2 D-cache, and the number of DTLB misses as measured by the Athlon performance monitoring counters. The graph shows that the programs with the largest speedups generally have dramatically reduced miss rates at every level of the cache hierarchy. The benefits for `twolf` and `llu-bench` are primarily at the TLB and those of `ft` are much greater in

the L1/L2 caches. For all other cases, the reductions are closely correlated at all the three levels of the memory hierarchy. This indicates that in these cases, the performance benefits are primarily due to smaller working sets, which would be produced by defragmenting the heap.

6.3.7 Access Pattern and Locality Changes

Section 6.3.4 shows that two programs, "ft" (from the Ptrdist suite) and "chomp" (from the McGill suite), speed up by over 10x with automatic pool allocation, and Section 6.3.6 shows that this is due to a dramatic reduction in cache and TLB misses. To characterize the source of this effect, we study these two programs in detail. We find that in both cases, the source of this dramatic speedup is due to success at our stated goals: segregating distinct data structures in memory from each other.

To evaluate the performance behavior of the program, we instrument the programs to capture a trace of all of the dynamic loads that the program executes. Given this data, we filter out accesses that are not directed to the heap, number the remaining loads in order, and plot the address loaded vs the load number. This generates a plot like that shown in Figure 6.15. We choose to not plot data for stores, as loads are typically the primary performance problem for heap-intensive programs (a load that misses in the cache blocks all instructions with true dependences on the load from executing).

For this study, we generate two plots: one when the program is running with standard malloc, and one when the program is changed to use the pool allocator. In both cases, we color the load on the plot to indicate which pool the load would target if the program were pool allocated (to make it easier to correlate data between the charts). Because the dynamic loads executed do not depend on the memory allocation pattern, the X axis of the charts match each other exactly. The Y axes, on the other hand, depends on the address in memory that the allocator placed the object being loaded from.

In runs that use the pool allocator, we disallow the system malloc implementation from calling mmap to satisfy allocation requests. The pool allocation runtime library requests (relatively) large blocks of memory from the system malloc to implement its internal memory slabs (described in Section 5.1.1). Without this tweak, the first several pool slabs are allocated in with the 'brk'

system call, then the later ones are allocated with mmap. While there should be no substantial performance difference between using sometimes using mmap and always using 'brk', it makes the graphs much harder to visualize: the giant address range difference between the mmap region and the brk region dwarfs the address range differences within either region (compressing both regions to horizontal lines). Note that runs with the standard malloc library do not need this tweak because they only allocate relatively small objects of less than 1000 bytes.

Finally, note that these figures are most easily understood in color. If possible, obtain the color postscript or PDF version of this thesis from the LLVM web site to see them.

Impact of Automatic Pool Allocation on chomp

chomp is an solver for a simple two player board game, configured to play against itself. chomp allocates three different nodes, which we call L, P, and D. L is an 8-byte object of type `_list`, P is a 16-byte object of type `_play`, and D is an array of int. To explain how the pool allocator is able to realize a 10x speedup on chomp, Figures 6.14 and 6.15 plot every load in chomp that accesses the heap, using malloc and the pool allocator, respectively. In these charts, the L objects are green, the P objects are red, and the D objects are blue[1]. In this (reduced) execution of chomp, we see that it has three phases: construction, processing and destruction, and that the processing phase makes eight distinct traversals over the L and P lists (corresponding to the vertical lines in Figure 6.14).

Like many programs, chomp uses an irregular allocation pattern, and generally intermixes object allocations (e.g. it starts with DPDLDPDLDLDDDPDLDL...). When using malloc, these objects are interspersed on the heap, roughly corresponding to allocation order (reuse of freed memory makes it inexact). When using the pool allocator, the three different objects are put in separate pools, and objects in each pool end up roughly in allocation order (P is exactly in allocation order, because nodes are never freed to its pool). The D objects are relatively large (compared to the L and P nodes) but are not accessed very frequently.

These allocation/layout patterns mean that, without Pool Allocation, the L and P list nodes are dispersed in memory (e.g. with variable strides of 100-500 bytes for the P objects) whereas the

[1]Note that our data plotting tool cannot draw data points transparently. As such, in Figure 6.14, all of the red points are covered with green points during the processing phase, and the red/green points are covered by the blue points during the construction and destruction phases. All of these points are easily visible in Figure 6.15.

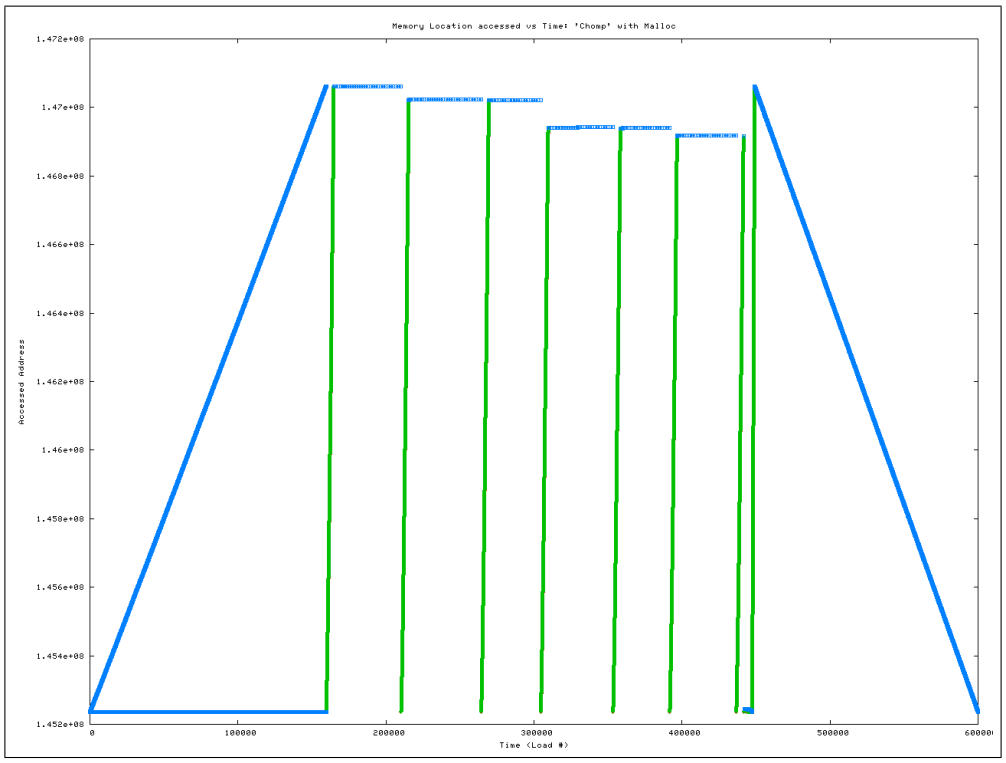


Figure 6.14: chomp Access Pattern with Standard malloc/free

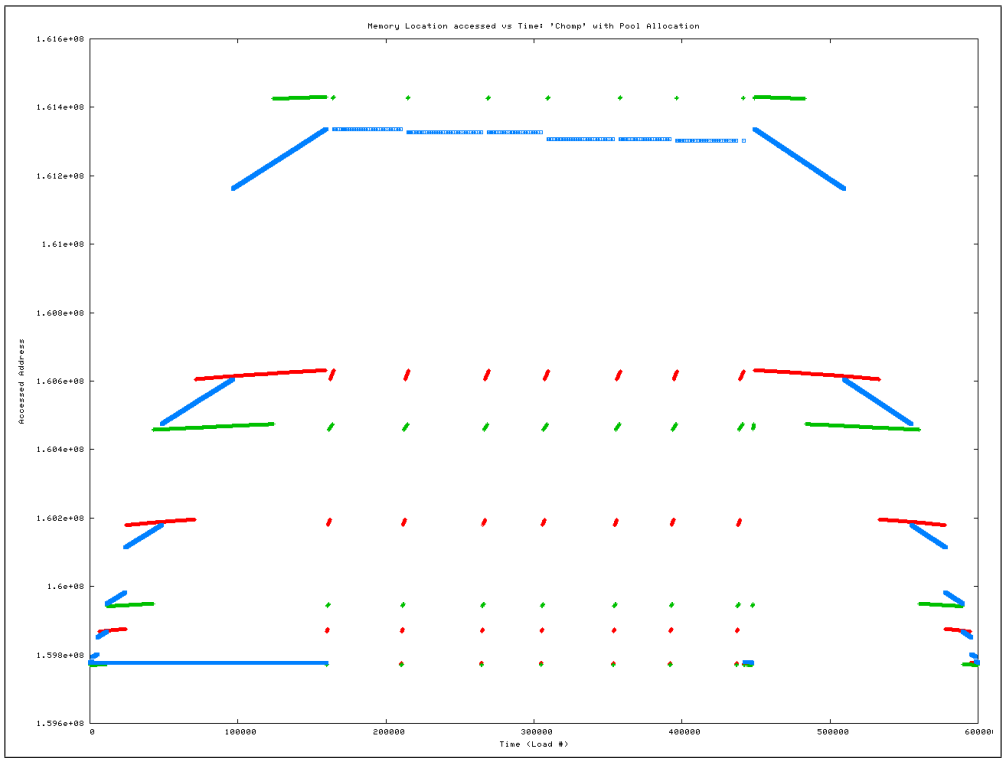


Figure 6.15: chomp Access Pattern with Pool Allocation

pool allocator packs them together, achieving a perfect stride of 20 bytes for the P objects (16 for the object and 4 for the object header). In Figure 6.14, we can see that the traversals of these L and P lists pulls most of the heap into and out of cache, and because the nodes have D allocations interspersed between them (which are not used in these traversals), each cache line fetch has at most one useful 8 or 16-byte object on it (which is much smaller than the cache line), wasting cache capacity and memory bandwidth on unused D objects.

Figure 6.15 shows that pool allocation segregates the linked lists from each other (and from the D objects), allowing these traversals to cover a much smaller address range. Because these small objects are packed densely together, a cache line fetch for one object will bring other useful objects on the same cache line into the cache at the same time. This improves cache density and reduces memory bandwidth required. The figure also shows the behavior of the pool allocation runtime library (described in Section 5.1.1), where it allocates chunks of memory from malloc to hold pool objects, doubling the size of the chunks each time it fills a chunk.

This change dramatically reduces the cache footprint of linked list traversals over the P and L nodes. In the case of the P list, it yields optimal cache density and provides the hardware stride prefetcher with a linear access pattern. This combination provides a reduction from 251M L1 misses to 63M L1 misses. Also, because the range of accessed memory is much smaller for these traversals, TLB misses are greatly reduced.

Impact of Automatic Pool Allocation on "ft"

The ft program (from the Ptrdist benchmark suite [8]) is an implementation of the minimum spanning tree algorithm described in [57]. It first creates a random undirected graph, then computes the minimum spanning tree of it. The input used for the performance numbers above (e.g., Section 6.3.4) builds a graph with 6000 nodes and 100,000 edges, which is large, but not unreasonably so. Figures 6.16 and 6.17 show the access pattern of a small input to ft (30 nodes and 150 edges) with malloc (the former) and with Automatic Pool Allocation (the later). We show a reduced input to make it easier to understand the figures.

ft consists of four main phases. The first phase creates the nodes for the random graph, the second phase adds the random edges to the graph, the third phase computes the minimum spanning

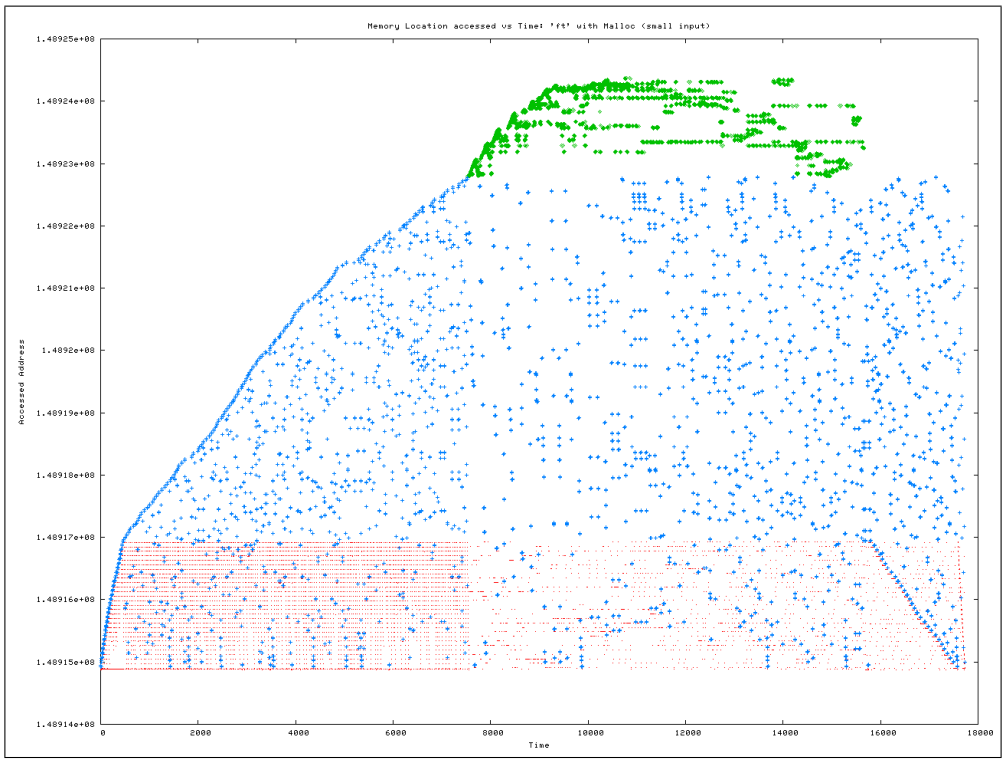


Figure 6.16: ft Access Pattern with Standard malloc/free

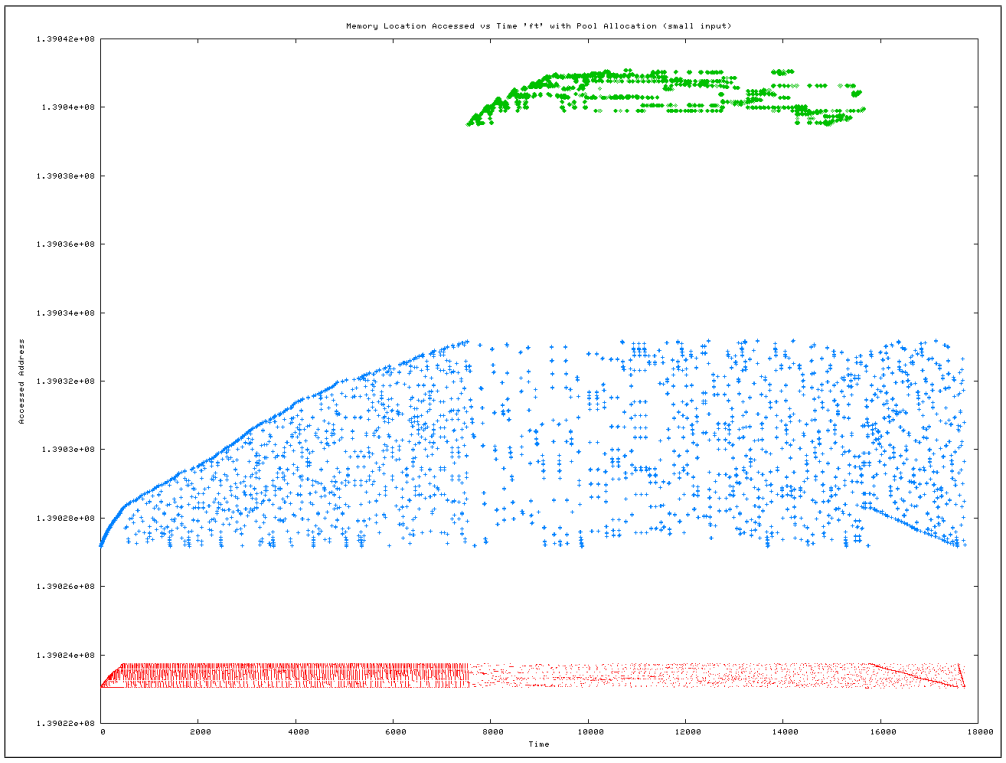


Figure 6.17: ft Access Pattern with Pool Allocation

tree, and the fourth phase prints out information about the spanning tree and the graph. The first phase is roughly from load #0 to #400, the second from #400 to #7600, the third from #7600 to #15750 and the last is from #15750 on.

ft allocates three different types of heap objects, which we name V, E, and H. V is a 20-byte structure of type `Vertices`, used to represent each vertex in the graph. E is a 16-byte structure of type `Edges`, used to represent each edge in the graph. H is a 28-byte object of type `Heap`, which represents the fibonacci heap used to solve the spanning tree problem. In the figures, the V nodes are red dots, the E nodes are blue crosses, and the H nodes are green diamonds.

The figures show that Automatic Pool Allocation is able to segregate the nodes of each of these different data structures into different memory spaces, whereas malloc interlaces the V and initial E nodes. In particular, when allocating the nodes for the graph (the first phase), the program allocates two edges for every node that it inserts into the graph (producing a pattern of VEEVEEVEEVEEVEE...). This pattern results in each of the V objects having two E objects between them, separating them in memory by 72 bytes: 20 bytes for the V node, a 4 byte object header, two 16-byte E nodes, and two 8-byte headers for the E objects (one word of alignment padding one word of object header. With the pool allocator, the nodes are allocated seperately from each other, with a 24 bytes offset between the nodes. When both the bump pointer and alignment optimization are enabled[2], the offset between these nodes shrinks to 20 bytes because the nodes are never free'd and the nodes contain no data that requires 8-byte alignment.

The ft program makes many traversals over the list of nodes during Phase 2 (easily visible in the bottom left of Figure 6.17), thus this dense packing of node objects allows each cache line fetch of a V node to pull other V nodes into the cache (instead of unrelated E nodes). This improves effective cache density, reduces memory bandwidth requirements, and reduces the working set for the program. This also makes it more likely that the V nodes will fit in the cache during Phase 3. The combination of bump-pointer and alignment optimizations further improve this, speeding up ft by about 18% over base the pool allocation performance. An optimization like instance interleaving [136] (discussed briefly in Section 8.2.1), would improve performance even more by

[2]If the bump-pointer optimization is disabled, 4 bytes are required for an object header. If the alignment optimization is disabled, the four bytes saved by the bump-pointer optimization are replaced with 4 bytes of alignment padding.

improving the density of the 'next' field accesses in the V structure.

After pool allocation and optimizations, the V list is often traversed with a constant (backward) stride of 20 bytes (because the newly allocated nodes are added to the front of the list). Because the nodes fit on fewer hardware pages, the automatic stride prefetcher on the Athlon is slightly more effective: with malloc it prefetches with a (backward) stride of 72 bytes (prefetching 56 nodes per page); with pool allocation it prefetches with a (backward) stride of 20 bytes (prefetching 204 nodes per page). This effect is significant, because the Athlon prefetcher stops on virtual memory page boundaries [80].

In addition to optimizing the V list, the pool optimizations apply to both other node types, though they do not contribute significantly to the performance improvement of FullPA over BasePA. The H structures are eligible for the alignment optimization but not the bump pointer optimization (nodes are freed and reallocated to the fibonacci heap), but this optimization does not eliminate any inter-node padding in this case (because the H nodes are of size $8n + 4$ with $n = 3$). The E list is eligible for both the bump pointer and alignment optimization, which reduces inter-node padding from 8 bytes to 0 bytes. Because accesses to the edge list suffer from poor locality even after pool allocation (neighboring nodes in memory are seldom accessed together), this improvement to the E list does not significantly improve program performance, but may contribute to reduced TLB miss rates.

Overall, we see that segregation of the V and E lists, which is one of the main goals of Automatic Pool Allocation, greatly improves the performance of this program.

Summary of Automatic Pool Allocation Impact

While chomp and ft are extreme cases, they illustrates perfectly how segregating and deinterlacing unrelated data structures can have a significant performance impact on heap-intensive program performance. Note that Automatic Pool Allocation may have an even more significant effect for real-world programs which fragment their heap over time: If the heap starts out fragmented, even linear allocations of memory (without interspersed allocations of other node types) may be fragmented in the heap. With Automatic Pool Allocation, these nodes are more likely to be grouped coherently together.

6.4 Research Contributions of Pool Allocation Optimizations

This chapter makes the following major research contributions beyond those described in Section 5.7:

(i) We present several simple but novel optimizations that can substantially improve the performance of heap data structures on a per-pool basis, beyond what pool allocation already provides. Several of these optimizations are general enough that they could be reimplemented in other existing region inference implementations.

(ii) We present an extension of the Automatic Pool Allocation transformation to support node collocation, present several example collocation heuristics, and describe our experiences with collocation.

(iii) We provide detailed performance results for both the Automatic Pool Allocation transformation itself and the pool optimizations presented in this chapter. We show that Automatic Pool Allocation and its optimizations improves the performance of several programs by 10-20%, speeds up two by about 2x and two others by about 10x. We show the locality is substantially improved by these transformations.

(iv) We use the chomp and ft benchmarks to show graphically how the Automatic Pool Allocation is meeting its goal of segregating distinct data structures – which dramatically improves the performance of these codes by more than 10x. We analyze and describe exactly what happens and how it relates at the source level.

Finally, we note again that pool allocation can be used as the basis for subsequent optimizations and analysis. Chapter 7 describes an aggressive macroscopic optimization (Transparent Pointer Compression) and Chapter 8 describes several non-performance-related applications of pool allocation.

Chapter 7

Transparent Pointer Compression

64-bit computing is becoming increasingly important for modern applications. Large virtual address spaces are important for several reasons, including increasing physical memory capacity, rapidly growing data sets, and several advanced programming techniques [121, 145, 70].

One problem with 64-bit address spaces is that 64-bit pointers can significantly reduce memory system performance [98] compared to 32-bit pointers[1]. In particular, pointer-intensive programs on a 64-bit system will suffer from (effectively) reduced cache/TLB capacity and memory bandwith for the system, compared to an otherwise identical 32-bit system. The increasing popularity of object oriented programming (which tends to be pointer intensive) amplify the potential problem. We observe that the primary use of pointers in many programs is to traverse linked data structures, and veryfew *individual* data structures use more than 4GB of memory, even on a 64-bit system. The question therefore is: How can we use pointers more efficiently to index into individual data structures?

This chapter[2] presents a sophisticated compiler transformation, Transparent Pointer Compression for Linked Data Structures, which automatically compresses pointers in type-safe data-

[1]Thanks to Wen-mei Hwu's research group for bringing this issue to our attention.
[2]Note that an updated version of this content will be published in [90].

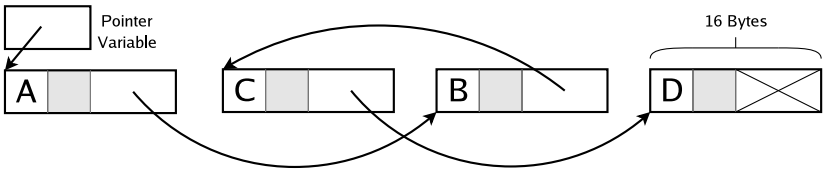


Figure 7.1: Linked List of 4-byte characters

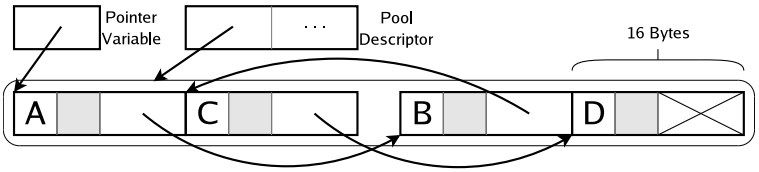


Figure 7.2: Pool Allocated Linked List

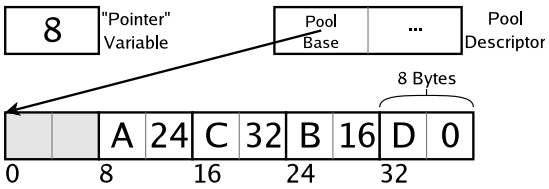


Figure 7.3: Pointer Compressed Linked List

structures (e.g. from 64-bits to 32-bits or less), while conservatively leaving non-type-safe data structures unmodified. Transparent Pointer Compression first pool allocates (Chapter 5) the code, then compresses pointers by replacing 64-bit pointers with smaller integer indexes from the start of these pools.

Consider a simple linked list of integers. Figure 7.1 illustrates the list compiled without pointer compression, and Figure 7.3 illustrates the memory organization with pointers compressed to 32-bit integer indexes. In this example, each node of the list originally required 16 bytes of memory[3] (4 bytes for the integer, 4 bytes of alignment padding, and 8 bytes for the pointer), and the nodes may be scattered throughout the heap. In this (extreme) example, pointer compression reduces each node to 8 bytes of memory (4 for the integer, and 4 for the index that replaces the pointer). Each index holds the offset of the target node from the start of the pool instead of an absolute address in memory.

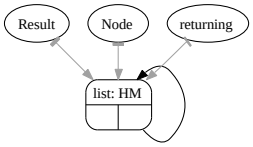
This chapter is organized as follows. In Section 7.1, we first describe a "static" version of pointer compression which limits individual pools to 2^k bytes each, for some $k < 64$ (e.g., $k = 32$). Section 7.2 extends this basic approach with a "dynamic" scheme that speculates that pointers will be small (and thus shrinks them) but allows them to dynamically expand if full addressing generality is required. Section 7.3 describes important optimizations over the basic algorithm required to achieve good performance of the generated code. Section 7.4 evaluates the performance impact and memory usage impact of the static form of pointer compression, Section 7.5 contrasts

[3]Not including overhead added by malloc.

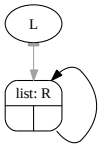
this work to previous work, and Section 7.6 concludes the chapter.

```
struct list { int X; list *Next; };

list *MakeList(int N) {
  list *Result = 0;
  for (int i = 0; i != N; ++i) {
    list *Node =
       malloc(sizeof(list));
    Node->Next = Result;
    Node->X = i+'A';
    Result = Node;
  }
  return Result;
}
int Length(list *L) {
  if (L == 0) return 0;
  return Length(L->Next)+1;
}
int Testlists() {
  list *A = MakeList(100);
  list *B = MakeList(20);
  int Sum = Length(A) + Length(B);
  ((char*)B)[5] = 'c'; // not type safe!
  return Sum;
}
```

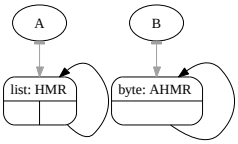
(a) Original



(b) BU DSA graph for MakeList



(c) BU DSA graph for Length



(d) BU DSA graph for Testlists

```
struct list { int X; list *Next; };

list *MakeList(Pool *PD, int N) {
  list *Result = 0;
  for (int i = 0; i != N; ++i) {
    list *Node =
       poolalloc(PD, sizeof(list));
    Node->Next = Result;
    Node->X = i+'A';
    Result = Node;
  }
  return Result;
}
```

```
int Length(list *L) {
  if (L == 0) return 0;
  return Length(L->Next)+1;
}
int Testlists() {
  Pool P1, P2;
  poolinit(&P1, sizeof(list));
  poolinit(&P2, 0 /*no size hint known*/);
  list *A = MakeList(&P1, 100);
  list *B = MakeList(&P2, 20);
  int Sum = Length(A) + Length(B);
  ((char*)B)[5] = 'c';
  pooldestroy(&P1); pooldestroy(&P2);
  return Sum;
}
```

(e) After Pool Allocation

Figure 7.4: Simple linked list example

7.1 Static Pointer Compression

Static pointer compression reduces the size of pointers in data structures in two steps. First, it replaces pointers in data structures with integers representing offsets from a pool base (i.e., indexes into the pool). Second, in order to compress this index, it attempts to select an integer type that

```
struct list_pc32 { int X; int Next; };

static int MakeList_pc32(Pool *PD, int N) {
  int Result = 0;
  for (int i = 0; i != N; ++i) {
    int Node = poolalloc_pc(PD, 1);
    int *tmp1 = PD->poolbase+Node+offsetof(list_pc32 , Next);
    *tmp1 = Result;
    int *tmp2 = PD->poolbase+Node+offsetof(list_pc32 , X);
    *tmp2 = i+'A';
    Result = Node;
  }
  return Result;
}

static int Length_pc32(Pool *PD, int L) {
  if (L == 0) return 0;
  int *tmp = PD->poolbase+L+offsetof(list_pc32 , Next);
  return Length_pc32(PD, *tmp)+1;
}

int Testlists() {
  Pool P1, P2;
  poolinit_pc(&P1, sizeof(list_pc32));
  poolinit_pc(&P2, 1);
  int A = MakeList_pc32(&P1, 100);
  int B = MakeList_pc64(&P2, 20);
  int Sum = Length_pc32(&P1, A) + Length_pc64(&P2, B);
  ((char*)B)[5] = 'c';
  pooldestroy_pc(&P1);
  pooldestroy_pc(&P2);
  return Sum;
}
```

Figure 7.5: Example after static compression

is smaller than the pointer size (e.g. by using a 32-bit integer on a 64-bit host). We refer to these as "index conversion" and "index compression" respectively. The latter step may fail because it requires somewhat stronger safety guarantees; nevertheless, we still perform the first step to achieve uniform code sequences for accessing compressed and uncompressed pools[4]. Static pointer compression will cause a runtime error if the program allocates more than 2^k bytes from a single pool using k-bit indices. Techniques to deal with this in the static case are discussed briefly in Section 7.1.4. Alternatively, this problem is solved by the dynamic algorithm in Section 7.2, but that algorithm is more restrictive in its applicability.

For our list example of Figure 7.4(a), the static pointer compression transformation transforms the code to that in Figure 7.5. Pointers to the `A` list are index-converted and compressed whereas

[4]Note that index conversion alone may also be useful for purposes other than pointer compression because it provides "position independent" data structures that can be relocated in memory *without rewriting any pointers* other than the pool base.

those to the `B` list are converted but not compressed, for reasons explained below. This also requires that distinct function bodies be used for the `A` and `B` lists (those for the former are shown). By shrinking pointers from 64-bits to 32-bits (which also reduces intra-object padding for alignment constraints), each object of the `A` list is reduced from 16 to 8 bytes – effectively reducing the cache footprint and bandwidth requirement by half for these nodes. The dynamic memory layout of the `A` list is transformed from that of Figure 7.2 to Figure 7.3.

To simplify the presentation, we describe the transformation in three pieces. First we describe changes required to the pool allocation runtime library to support pointer compression. Next, we describe the transformation for data structures that are never passed to or returned from functions, intraprocedural static pointer compression (Section 7.1.2). Finally we describe the approach to handle function calls (Section 7.1.3).

7.1.1 Pointer Compression Runtime Library

The pointer compression runtime library is almost identical to the standard pool allocator runtime described in Section 5.1.1. The only two functionality differences are that it guarantees that the pool is always contiguous (realloc'ing the entire pool to grow it, or using the technique described in Section 7.3.1) and that it reserves the 0^{th} node to represent the null pointer. The library interface is also cosmetically different in that the memory allocation/free functions take indices instead of pointers, and numbers of nodes to allocate instead of number of bytes. The API is listed Figure 7.6.

```
void poolinit_pc(Pool* PP, unsigned NodeSize);
    Initialize the pool descriptor, record node size.
void pooldestroy_pc(Pool* PP)
    Release pool memory and destroy pool descriptor.
int poolalloc_pc(Pool* PP, uint NumNodes)
    Allocate NumNodes nodes.
void poolfree_pc(Pool* PP, int NodeIdx)
    Mark the nodes starting at NodeIdx as free.
void* poolrealloc_pc(Pool* PP, int NodeIdx ptr, uint
NumNodes)
    Resize an object to NumNodes nodes.
```

Figure 7.6: Pool Compression Runtime Library

7.1.2 Intraprocedural Pointer Compression

Given the points-to graph and the results of automatic pool allocation, intraprocedural static pointer compression is relatively straight-forward. The high level algorithm is shown in Figure 7.7. Each function in the program is inspected for pools created by the pool allocator. If index-conversion is safe for such a pool, any instructions in the function that use a pointer to objects in that pool are rewritten to use indexes off the pool base. Indexes in memory are stored in compressed form (k bits) when safe, otherwise left in uncompressed form (i.e., 0-extended to 64 bits). The pool is also marked to limit its aggregate size to 2^k bytes.

```
   pointercompress(program P)
1    poolallocate(P)                    // First, run pool allocator
2    ∀F ∈ functions(P)
3      set PoolsToIndex = ∅
4      ∀p ∈ pools(F)                    // Find all pools
5        if (safetoindex(p))            // index-conversion safe for p?
6          PoolsToIndex = PoolsToIndex ∪ {p}
7      if (PoolsToIndex ≠ ∅)
8        rewritefunction(F, PoolsToIndex)
```

Figure 7.7: Pseudo code for pointer compression

The $safetoindex$ predicate used on line #5 controls what pools are considered safe to access via indexes instead of pointers. For intraprocedural pointer compression, the constraints are:

1. The pool lifetime must be bounded by this function.

2. The points-to graph node corresponding to the pool must represent only heap objects and no other class of memory (i.e., no global or stack objects).

3. The pool cannot be passed into a function call.

Constraint #1 is directly identified by the pool allocator. If the constraint is not satisfied, it may still be index-converted in some parent function (via the full interprocedural algorithm described below). Constraint #2 is determined from the points-to graph produced by DSA. It is required because stack and global data are not be allocated out of a heap pool, and pointers to such objects cannot easily be converted into offsets relative to the base of such a pool. Constraint #3 is also identified by DSA, and is relaxed in Section 7.1.3. In practice, we impose a profitability constraint

as well: we only apply convert pools that are pointed to by heap memory objects. If a heap object is not pointed to by any memory object (including itself), no pointers in memory will be shrunk by indexing the pool, so there is no reason to index-convert it.

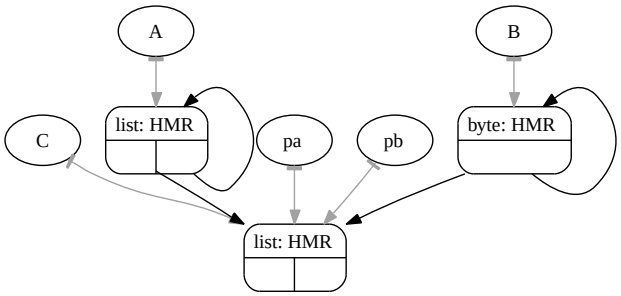


Figure 7.8: Example with TH and non-TH nodes
Pointers from list A to list C can use compressed indices; those from list B to list C must use uncompressed indices.

Once the indexable pools have been identified in the function, these pools will be used to hold at most 2^k bytes, $k < n$, where n is the pointer size for the target architecture (e.g., $k = 32$ and $n = 64$). All valid pointers into such pools are replaced with indexes in the range $[1 \ldots 2^k - 1]$. Some of these index variables, however, must still use a full n bit representation (i.e., 0-extended from k to n bits) if, for example, the compiler cannot safely change the layout of an object containing the variable. By definition, objects represented by TH nodes of the points-to graph (see Chapter 3) can be safely reorganized; index values in such objects are stored using k-bits. For example, in Figure 7.4, the A list objects can be reorganized and therefore can hold compressed indices whereas the B list objects cannot (this would still be true if both lists pointed to a common indexed pool).

For example, Figure 7.8 shows a points-to graph in which a node (list C) is pointed to by a TH node (list A) and a non-TH node (list B). The pool for the C lists can be index-converted, the pointers from the A list to the C lists can be compressed to k-bit indices, but those from the B lists to the C list must be recorded as n-bit indices. Assume the scalar pointer variables pa and pb are loaded out of the A and B lists (e.g., pa = A->next->val and pb = B->next->val). Then, pa and pb will both hold n-bit values, but different code sequences must be used for these two loads.

Once the indexable pools and the compressible index variables have been identified in the function, a single linear scan over the function is used to rewrite instructions that address the indexable pools. Assuming a simple C-like representation of the code which has been lowered to

individual operations, the rewrite rules are shown in Figure 7.9 (operations not shown here are unmodified).

Original Statement		Transformed Statement
P = null	$\Rightarrow$	$P' = 0$
$P_1 = P_2$	$\Rightarrow$	$P'_1 = P'_2$
cc = $P_1 \stackrel{?}{=} P_2$	$\Rightarrow$	cc = $P'_1 \stackrel{?}{=} P'_2$
$P_1 = \&P_2\text{->}field$	$\Rightarrow$	$P'_1 = P'_2$ + newoffsetof($field$)
$P_1 = \&P_2[V]$	$\Rightarrow$	$P'_1 = P'_2$ + V*newsizeof($P_2[0]$)
If node(P) is non-TH or τ not a pointer ($P : \tau*$):		
V = *(τ*)P	$\Rightarrow$	Base = PD->PoolBase V = *(τ*)(Base+P')
((τ*)P) = V	$\Rightarrow$	Base = PD->PoolBase *(τ*)(Base+P') = V
If node(P) is TH and τ is a pointer ($P : \tau*$):		
P_1 = *P	$\Rightarrow$	Base = PD->PoolBase P'_1 = *($IdxType$*)(Base+P')
$P_1 = P$	$\Rightarrow$	Base = PD->PoolBase *($IdxType$*)(Base+P'_1) = P'
P = poolalloc(PD, N)	$\Rightarrow$	Tmp = N/OldSize P' = poolalloc_pc(PD, Tmp)
poolfree(PD, P)	$\Rightarrow$	poolfree_pc(PD, P')
poolinit(PD, Size)	$\Rightarrow$	Tmp = Size/OldSize*NewSize poolinit_pc(PD, Tmp)
pooldestroy(PD)	$\Rightarrow$	pooldestroy_pc(PD)

Figure 7.9: Rewrite rules for pointer compression

In the rewrite rules, "P" and "P'" denote an original pointer and a compressed index. "V" is any non-compressed value in the program (a non-pointer value, a non-converted pointer, or an uncompressed index). "$IdxType$" is the integer type used for compressed pointers (e.g. int32_t on a 64-bit system). All P' values are of type $IdxType$. Indexes loaded from (or stored to) non-TH pools are left in their original size whereas those from TH pools are cast to $IdxType$.

The rules to rewrite addressing of structures and arrays lower addressing to explicit arithmetic, and use new offsets and sizes for the compressed objects, not the original. Memory allocations scale (at runtime) the allocated size from the old to the new size. The most common argument to a `poolalloc` call is a constant that is exactly "OldSize", allowing the arithmetic to constant fold to NewSize. The dynamic instructions are only needed when allocating an array of elements from a single `poolalloc` site, or when a malloc wrapper is used (in the interprocedural case).

7.1.3 Interprocedural Pointer Compression

Extending pointer compression to support function calls and returns requires four changes to the algorithm above. First, constraint #3 from $safetoindex$ is eliminated. Second, a minor change is needed to the pool allocation transformation to pass pool descriptors for all pools accessed in a callee (or it's callees), not just those pools used for `malloc` or `free` in the callee (this is accomplished by removing the check from Line #4 of Figure 5.4).

In Figure 7.4 for example, the `Length` function now gets a pool descriptor argument for "`L`." Third, the rewrite rules in Figure 7.10 must be used to rewrite function calls and returns. Fourth, and most significantly, interprocedural pointer compression must handle the problem that a reference in a function may use either compressed or non-compressed indices in different calling contexts.

Original Statement		Transformed Statement
$P_1 = F(P_2, \mathrm{V}, P_3, ...)$	$\Rightarrow$	$P_1' = F_c(P_2', \mathrm{V}, P_3', ...)$
$\mathrm{V}_1 = F(\mathrm{V}_2, P_2, ...)$	$\Rightarrow$	$\mathrm{V}_1 = F_c(\mathrm{V}_2, P_2', ...)$
$F(\mathrm{V}_1, \mathrm{V}_2, ...)$	$\Rightarrow$	$F(\mathrm{V}_1, \mathrm{V}_2, ...)$
return P	$\Rightarrow$	return P'

Figure 7.10: Interprocedural rewrite rules.
Pool descriptor args. added by pool allocation are not shown. They are ignored during pointer compression.

The fourth problem arises because the same points-to graph node in a callee function can correspond to different pools in different calling contexts. One context may pass a TH pool and another a non-TH pool, requiring different code to load or store pointers in these two pools. We propose two possible solutions to this problem. The first is to generate conditional code for loads and stores of such index values (*uses* of these indexes are not a concern because they are always used as n-bit values). The second is to use function cloning and generate efficient, unconditional code in each function body. As explained in the next section, dynamic pointer compression *requires* conditional code sequences in any case to handle dynamic pool expansion, and we describe the former solution there. Our goal with static pointer compression is to present a very efficient solution that works in most common cases, and therefore we focus on the latter solution (function cloning) here. In practice, we believe that relatively little cloning would be needed for many programs.

Figure 7.4 shows a case when cloning must be used. In particular, `Testlists` in Figure 7.4 calls `MakeList` and `Length` and passes or gets back data from indexed pools into each of them.

Since the A list indices are compressed but the B list ones are not, the transformation needs to create two versions of `MakeList` and `Length`, one for each case. The A list version (denoted by suffix "_pc32") is shown; the second version is the same except it uses the uncompressed rewrite rules for loads and stores of pointers in Figure 7.10. Only two versions are needed for each function because only one pool within each function (the `list` node) is accessed in multiple ways. In the worst case, cloning can create an exponential number of clones for a function: one clone for each combination of compressed or uncompressed pools passed to a function. In practice, however, we find that we rarely encounter cases where TH and non-TH pools containing heap objects point to a common indexed pool or are passed to the same function.

Given the extensions described above, interprocedural static pointer compression is a top-down traversal of the program call graph, starting in main and cloning or rewriting existing function bodies as needed. Our implementation of static pointer compression does not support indirect function calls, so the single static callee is always for each call site. All together, applied to the example in Figure 7.4, static pointer compression produces the code in Figure 7.5.

7.1.4 Minimizing Pool Size Violations with Static Compression

Static compression is not a completely safe transformation because a correct program may fail if it tries to allocate more than 2^k bytes from a pool that uses k-bit indices. Nevertheless, we believe this transformation can be used safely in practice on many programs. First, each pool only holds a single instance of a data structure instance or even a subset of an instance (if the data structure consists of multiple nodes in the points-to graph). This means that part or all of a *single* DS instance must exceed 2^k bytes (e.g., 4GB for k=32) before an error occurs.

Second, many pools can be indexed by *objects* instead of *bytes*, thus expanding the effective maximum pool size greatly[5]. Node indexing can be used for TH pools holding objects for which the address of a field is not taken (i.e., all pointers point to the start of pool objects). This criterion is met by many data objects in C and C++ programs, and all those in Java programs.

Third, a compiler could use profiling runs (and simple runtime pool statistics) to identify pool instances that grow unusually large compared with other pools in a program and simply disable

[5]Node-indexing is actually required for dynamic compression, and is described below.

pointer compression for those pools. Finally, programmers could use options or #pragmas to specify that pools created in certain functions should not undergo index compression.

7.2 Dynamic Pointer Compression

Dynamic pointer compression aims to allow a pool instance to grow beyond the limit of 2^k bytes (or 2^k objects) by expanding compressed indices transparently at run time. The technique has a higher runtime overhead, and cannot be used for all indexed pools in C and C++ programs (this is not a problem in Java programs).

There are several possible ways to implement dynamic pointer compression. To make it as simple as possible to grow pools at run time, we impose three restrictions on the optimization. First, we compress and expand indices within objects in a pool only if it meets the criteria for node-indexing mentioned above: it must be a TH pool and the address of a field is not taken. Second, we allow only two possible index sizes to be used for a pool: the initial k bits (e.g., 32) and the original pointer size, n = `intptr_t` (e.g,. 64). Third, for any pool of objects containing compressible indices, we allow only two choices: all are compressed or all are uncompressed. For example, in list A in Figure 7.8, either both index fields (the pointers to the C list and the back edge to the A list) are stored in compressed form or both are stored in uncompressed form (of course, list B would have to be TH for the transformation to apply).

Section 7.2.1 describes the modified rewrite rules for dynamic pointer compression, Section 7.2.2 describes changes to the runtime, and Section 7.2.3 describes the needed changes to the interprocedural transformation.

7.2.1 Intraprocedural Dynamic Compression

In the discussion below, we refer to a pool containing indices as *source pools*, since they are the sources of pointers into indexed pools. A source pool is often also an indexed pool because many linked data structures are recursive, e.g., the pool for list `list2` in Figure 7.13. In this example, the `int` pool is indexed but is not a source pool.

Intraprocedural dynamic pointer compression is largely the same as static compression but more complex load/store code sequences are needed for objects containing compressed indices,

since these indices may grow at run time. In each source pool, we store a boolean value, "*isComp*," which is set to true when objects in the pool hold compressed indices and false otherwise. A single boolean is sufficient by our third restriction above (all indices in an object are compressed or all are uncompressed). If a source pool is also an indexed pool, all index values pointing to the pool held in registers, globals, or stack locations are represented using the full n bits (the high bits are zero when $isComp = true$). Without this simplification, pointer compression would have to expand the indices in all such objects when the pool exceeded 2^k nodes. This is technically feasible for global and stack locations (using information from the points-to graph) but probably isn't worth the added implementation complexity.

Figure 7.12 shows the main[6] rewrite rules used for dynamic pointer compression. The transformed version of the `Length` function in the example is shown in Figure 7.11. Because we do not compress pool indexes if the address of a field is taken, the code for addressing the field and loading it is handled by one rule. The generated code differs from the static compression case in two ways: 1) both compressed and expanded cases must be handled; and 2) node-indexing rather than byte-indexing is used, i.e., the pool index is scaled by the node size before adding to `PoolBase`. For the former, a single branch on *isComp* is sufficient because we restricted source objects to have all compressed or all uncompressed indices: there are only two cases for each source pool, and the object size and field offsets are fixed and known at compile-time for each case[7].

```
/* Length with dynamic pointer compression (64->32 bits) */
static int Length(Pool *PD, long L) {
  if (L == 0) return 0;
  long Next = PD->isComp ? (long)*(int *)(PD->PoolBase + Node*8  + 4)
                         :          *(long*)(PD->PoolBase + Node*16 + 8);
  return Length(PD, Next)+1;
}
```

Figure 7.11: Example after dynamic compression

We can use node- rather than byte-indexing because of the first restriction, which disallows pointers into the middle of an object in a pool with compressed indices. Node-indexing is important

[6]We only show the rules for loads of structure fields. Stores are identical except for the final instruction, and array accesses are similar.

[7]Branch-free sequences are possible for loads and stores for many architectures, and can be tuned for many specific values of the constants. We omit the details here for lack of space.

Original Statement		Transformed Statement
P = null	⇒	P' = 0
$P_1 = P_2$	⇒	$P_1' = P_2'$
cc = $P_1 \stackrel{?}{=} P_2$	⇒	cc = $P_1' \stackrel{?}{=} P_2'$

$P_1 = P_2$->$field$	⇒	char *Ptr = PD->PoolBase
		if (PD->isComp) {
		Ptr += P_2'*newsizeof(*pooltype*)
		Ptr += newoffsetof($field$)
		P_1' = *(int32_t*)Ptr
		} else {
		Ptr += P_2'*oldsizeof(*pooltype*)
		Ptr += oldoffsetof($field$)
		P_1' = *(int64_t*)Ptr
		}

τV = P->$field$	⇒	char *Ptr = PD->PoolBase
		if (PD->isComp) {
		Ptr += P'*newsizeof(*pooltype*)
		Ptr += newoffsetof($field$)
		} else {
		Ptr += P'*oldsizeof(*pooltype*)
		Ptr += oldoffsetof($field$)
		}
		V = *(τ*)Ptr

P = poolalloc(PD, N)	⇒	Tmp = N/OldSize P' = poolalloc_pc(PD, Tmp)
poolfree(PD, P)	⇒	poolfree_pc(PD, P')
poolinit(PD, Size)	⇒	poolinit_pc(PD, &TypeDesc, PD_1, PD_2, ..., NULL)
pooldestroy(PD)	⇒	pooldestroy_pc(PD)

Figure 7.12: Dynamic pointer compression rules

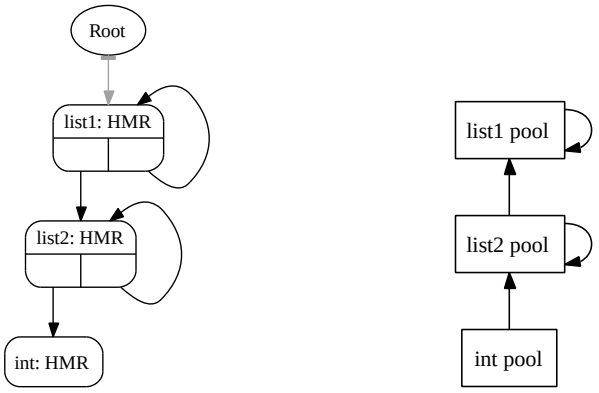
in order to limit which indices need to be rewritten at run-time when objects in a source pool are expanded (the specific run-time operations are described below). In particular, expanding objects in a pool does not change their node index, although it does change their byte offset. Therefore, when objects in a source pool are expanded (and the source pool itself is indexed), the node index values in the source pool do not need to change, only their sizes increase.

7.2.2 Dynamic Compression Runtime Library

The dynamic pointer compression runtime library is significantly different from the library for the static case. When a pool P grows beyond the 2^k limit, the run-time must be able to find and

expand (0-extend) all indices in all source pools pointing to this pool. This requires knowing which source pools point to pool P, and where the pointers lie within objects in these source pools.

To support these operations, the `poolinit` function takes static information about the program type for each pool (this type is unique since we only operate on TH pools), and is enhanced to build a *run time pool points-from graph* for the program. The type information for a pool consists of the type size and the offset of each pointer field in the type.



(a) Compile-Time Points To Graph (b) Run-Time Pool Descriptor Graph

Figure 7.13: Dynamic expansion example

The run time pool points-from graph has a node for each pool and an edge $P_2 \rightarrow P_1$, if there is an edge $N_1 \rightarrow N_2$ in the compiler's points-to graph, where P_1 and P_2 are the pools for nodes N_1 and N_2. An example points-to graph and the run-time points-from graph are shown in Figure 7.13(b). When `poolinit_pc` is called to initialize a pool descriptor (PD), it is passed some number of additional pool descriptor arguments ($PD_1 \ldots PD_n$). It adds PD to the "points-from" list of each descriptor $PD_1 \ldots PD_n$. For the example, when the `list2` pool descriptor is initialized, it is passed pointers to the `int` pool descriptor and itself (since the `list2` node has a self-loop), so it adds itself to the points-from lists in both pools. `pooldestroy_pc`(PD) removes the PD entry from $PD_1 \ldots PD_n$. The run-time points-from lists are created and emptied in this manner because, if $N_1 \rightarrow N_2$ in the compiler's points-to graph, then the lifetime of P_1 (for N_1) is properly nested within the lifetime of P_2 (for N_2).

At run time, if the $2^k th$ node is allocated from a pool, P, the "points-from" list in P is traversed, decompressing all the pointers in each pool in the list. For example, in Figure 7.13, when when the $2^k th$ node is allocated from the "`list2`" pool, both the `list2` pool and the `list1` pools need to be

decompressed so that all pointers into the `list2` pool are n-bit values. The normal metadata for a pool identifies which objects in the pool are live. *All* pointers in each live object are decompressed (because of our third restriction above). Decompressing each pointer simply means zero-extending it from k to n bytes. Decompression will grow the pool, which may require additional pages to be allocated and the pool base may change. As objects are copied to their new locations, their relative position in the pool is preserved so that all indices into the pool remain valid.

7.2.3 Interprocedural Dynamic Compression

As noted with static pointer compression, the primary challenge in the interprocedural case is that the same points-to graph node may represent pools containing compressed indices or non-compressed indices. This led to the possibility that functions must be cloned in the static case. Because dynamic pointer compression already uses conditional code to distinguish compressed indices from expanded indices, the need for cloning does not arise.

For interprocedural dynamic compression to compress indices in a pool, it must check if the pool meets the first criterion (TH pool, no field address taken) for all calling contexts. DSA computes two DS graphs for each function - a bottom-up (BU) graph representing a function and its callees (but not any callers), and a final, top-down (TD) graph representing the effects of both callees and callers. Therefore, we can check the criterion for all contexts trivially simply by checking it in the TD graph.

Original Statement		Transformed Statement
poolinit(PD, Size)	⇒	poolinit_pc(PD, NULL)
pooldestroy(PD)	⇒	pooldestroy_pc(PD)

Figure 7.14: Rewrite rules for non-compressed pools

Interprocedural dynamic pointer compression is very straight-forward: a single linear pass over the program is used to rewrite all of the instructions in the whole program, according to the rewrite rules in Figure 7.12 and Figure 7.14. The only difference between compressed and non-compressed pools (i.e., those that pass or fail the above criterion) is that the `poolinit_pc` call for the latter pool passes a null type descriptor (and an empty points-to list). In this case, `poolinit_pc` initializes the pool descriptor such that `PoolBase` is null and `isComp` is false, and ensure that the

`poolalloc_pc/free_pc` calls behave the same as `poolalloc/poolfree`.

This approach takes advantage of the fact that the pool allocator identifies data structures that do not escape from the program (the pool allocator cannot pool allocate something otherwise), which is the same legality constraint that dynamic pointer compression needs. Because `isComp` is false, non-compressed pools will always used the "expanded" code paths, which use the uncompressed sizes and field offsets for memory accesses.

7.3 Optimizing Pointer Compressed Code

The straight-forward pointer compression implementations described in Sections 7.1 and 7.2 generates functional, but slow, code. We describe several straightforward improvements below that can significantly reduce redundant or inefficent operations in the generated code.

7.3.1 Address Space Reservation

One of the biggest overheads of pointer compression is the need to keep the memory pools contiguous. If the pool allocator is built on top of a general memory allocator like `malloc`, growing the pool may require copying all its data to a new location with enough memory.

Given that this work targets 64-bit address space machines, however, a reasonable implementation approach is to choose a large static limit for individual data structures in the program that is unlikely to be exceeded (e.g., 2^{40}B), and reserve that much address space for each pool when it is created by the program (using facilities like `mmap(MAP_NORESERVE)`). This allows the program to grow a data structure up to that (large) size without ever needing to copy the pool, with the operating system kernel allocating memory pages to the data structure as they are used.

7.3.2 Reducing Redundant PoolBase Loads

Pointer compression requires loading the `PoolBase` and `isComp` fields from the pool descriptor for each load and store from a pool. Although these loads are likely to hit in the L1 cache, this overhead can dramatically impact tight pointer-chasing loops. Fortunately, almost all of these loads are redundant and can be removed with Partial Redundancy Elimination (or a combination of LICM and GCSE). The only operation that invalidates these fields is an allocation, either from

the pool (moving the pool base) or one of the pools it points to (decompressing pointers in the pool). The DS graphs directly identify which function calls may cause such operations.

Note that if Address Space Reservation is used, the `PoolBase` is never invalidated, making it reasonable to load it once into a register when the pool is initialized or in the prologue of a function if the pool descriptor is passed in as an argument. Figure 7.15 shows `MakeList_pc32` after simple optimizations on a 64-bit machine (assuming address space reservation is used).

```
static int MakeList_pc32(Pool *PD, int N) {
  char *PoolBase = PD->poolbase;
  int Result = 0;
  for (int i = 0; i != N; ++i) {
    int Node = poolalloc_pc(PD, sizeof(list_pc32));
    char *NodePtr = Poolbase+Node;
    *(int*)(NodePtr+4) = Result;
    *(int*)NodePtr = i+'A';
    Result = Node;
  }
  return Result;
}
```

Figure 7.15: `MakeList_pc32` after optimization

7.3.3 Reducing Dynamic `isComp` Comparisons

The generated code for dynamic pointer compression makes heavy use of conditional branches to test whether or not the pool is compressed. To get reasonable performance from the code, several standard techniques can be used. The most important of these is to use loop unswitching on small pointer chasing loops. This, combined with jump threading (merging of identical consecutive conditions) for straight-line code, can eliminate much of the gross inefficiency of the code, at a cost of increased code size. Other reasonable options are to move the "expanded" code to a cold section vs hot section, or use predication (e.g., on IA64).

7.3.4 Structure Field Reordering for Pointers

One of the overheads involved with dynamic pointer compression is that the offsets of fields are different in the compressed and uncompressed case. A reasonable way to help reduce this impact is to use structure field reordering to move all pointer fields to the end of the structure. After

performing this transformation, all offsets up to and including the first pointer field are constant and do not depend on "`isComp`".

7.3.5 Adding Hardware Support

Depending on the host ISA, several different forms of hardware support may be useful. For static compression, perhaps the most important hardware support is a "register+register+immediate" addressing mode (supported by the X86-64 ISA, for example).

For dynamic pointer compression, several options are possible. An integer multiply-accumulate instruction (which takes two immediates), coupled with a conditional move can be used to implement branch-free structure field indexing as:

```
reg1 = NodeIndex * newsize + newfieldoffset
reg2 = NodeIndex * oldsize + oldfieldoffset
reg2 = cmove isComp, reg1
load [reg2 + poolbase]
```

However, the most important operation to have is the ability to do either 32-bit or 64-bit loads (and stores) controlled by a predicate (e.g. "`reg = isComp ? LOAD32 [ptr] : LOAD64 [ptr]`')'. Architectures that support general predication (like IA-64) can do this with several instructions, but there is no good way to implement this without a branch on other systems (unless they support efficient unaligned 64-bit loads). This simple addition can make the unoptimizable case of dynamic compression much more efficient.

7.4 Experimental Results

We implemented the static approach to pointer compression in the LLVM Compiler Infrastructure (Chapter 2), building on our implementation of Data Structure Analysis (Chapter 3 and Automatic Pool Allocation 5. We use address space reservation to avoid reallocating pools and make redundancy elimination of PoolBase pointers easier (as described in Section 7.3). To evaluate the performance effect of Pointer Compression, we first look at how it affects a set of pointer-intensive

benchmarks, then investigate how the effect of the pointer compression transformation varies across four different 64-bit architectures.

7.4.1 Performance Results

Program	Native	PA	PC	PC/PA	PeakPA	PeakPC
bh	20.19	16.63	16.63	1.0	7.7MB	7.7MB
bisort	33.69	26.55	24.14	.909	48MB	24MB
perimeter	11.06	6.50	5.07	.780	256MB	149MB
power	12.56	6.80	6.78	.997	924KB	854KB
treeadd	73.10	53.57	35.86	.669	96MB	48MB
tsp	18.48	16.24	11.50	.708	131MB	114MB
ft	15.07	11.33	9.80	.865	8.9MB	4.5MB
ks	9.14	8.05	8.05	1.0	47KB	47KB
llubench	35.40	27.87	11.84	.425	3MB	1.5MB

Figure 7.16: Pointer Compression Benchmark Results

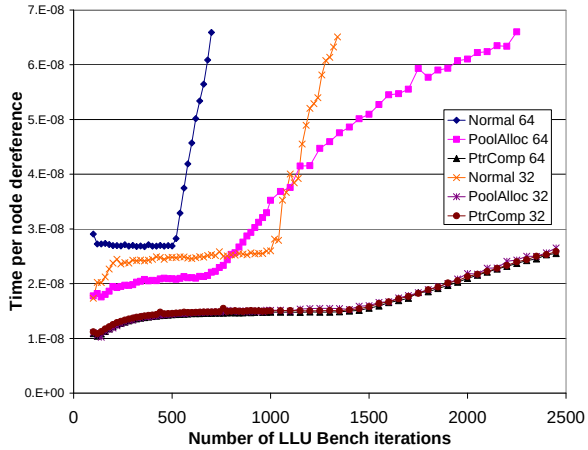
Figure 7.16 shows the results of using pointer compression on a collection of benchmarks running on a UltraSPARC-IIIi processor with 1MB of cache. The first column lists the benchmark name, which are drawn from the Olden [109], Ptrdist [8] and LLUbench [147] pointer intensive benchmark suites.

To evaluate the performance impact of pointer compression, we compiled each program with the LLVM compiler (including the pool allocation or pointer compression), emitted C code, and compiled it with the system GCC compiler. The PA and PC columns are the execution time for each benchmark with Pool Allocation or Pointer Compression turned on, and the PC/PA column is their runtime ratio (smaller is a bigger speedup). We include the runtime for the program, compiled just by GCC, in the 'Native' column to show that the pool allocated execution time for the program is a very aggressive baseline to compare against. Each number is the minimum of three runs of the program, reported in seconds.

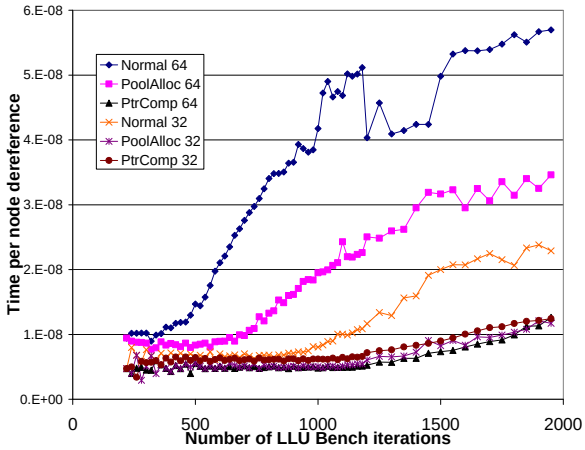
Pointer compression speeds up programs by over 2x in some cases (llubench) by dramatically reducing the cache footprint of the program. Even in cases that are less dramatic, pointer compression is able to speed up program by 20-30% over pool allocation. Some programs, however, are not helped. BH, for example, is not type-homogenous, so pointer compression does not compress anything. Power has such a small footprint that its main traversals are able to live in the cache,

even with 64-bit pointers. In KS, pointer compression shrunk a pointer, but the space saved is replaced by structure padding. Overall, as a program's memory image grows, the speedup provided by pointer compression should grow correspondingly.

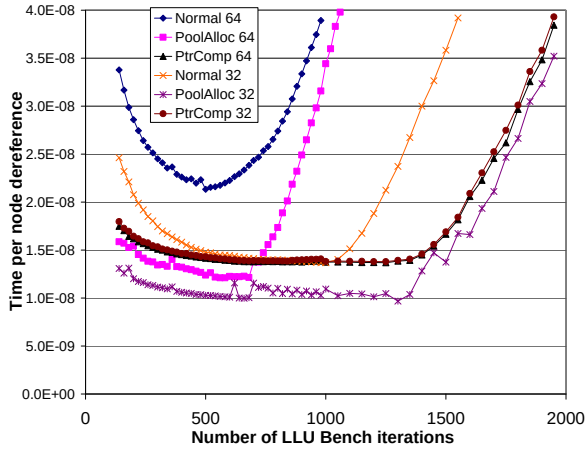
To show memory savings, we counted the peak number of bytes allocated by the program in pool allocated and pointer compressed forms. For these programs, pointer compression substantially reduces the heap image for the program as you would expect.



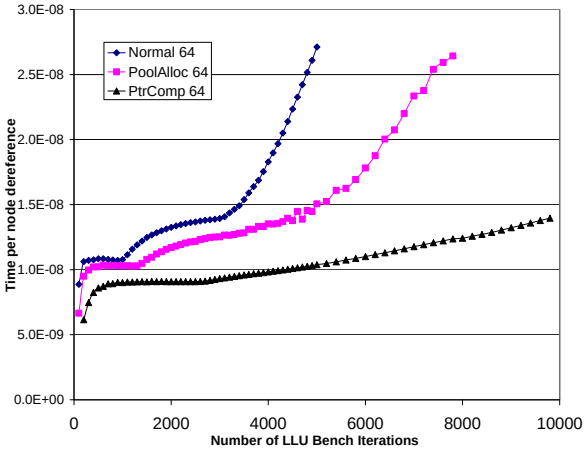
(a) 1Ghz UltraSPARC-IIIi - 1MB Cache



(b) 1.8Ghz AMD Opteron 244 - 1MB Cache



(c) 375Mhz IBM SP 9076-550 - 8MB Cache



(d) 1.6Ghz Itanium 2 Madison - 9MB Cache

Figure 7.17: llubenchmark: time to process one node vs problem size

7.4.2 Architecture Specific Impact of Pointer Compression

In order to evaluate the effect of pointer compression on different architectures, we chose to use a single bechmark, LLUbench, and a range of input sizes. We chose LLUbench, a linked-list microbenchmark, because its input size can be scaled over a wide range and it is small enough to get working on several platforms without having to port our entire compiler to each system.

Figure 7.4.1 shows the scaling behavior of llubench on four different systems, compiled in several configurations. For each configuration, we compiled and optimized the program using LLVM, emitted C code, then compiled the resultant code with a standard C compiler (IBM XLC for the SP, GCC for all others). We used 6 configurations for each platform: the original code (Normal), pool allocation only (PoolAlloc), and pointer compression (PtrComp), each compiled in 32-bit mode and in 64-bit mode (except the Linux Itanium system, which lacks 32-bit support).

The heap size used by llubench is a linear function of the number of iterations, but the execution time of the benchmark grows quadratically. To compare performance of different configurations and systems as a function of the heap size, therefore, we show the ratio of total running time to number of list nodes on the Y axis. This number increases with heap size because the processor spends more time stalled for cache misses[8].

Overall, 64-bit pointers have a major performance overhead compared to 32-bit pointers for all systems, when using either the native (Normal) or pool allocator. With a particular pointer size, the Automatic Pool Allocation transformation consistently increases locality over using the standard system allocator, particularly with the default Solaris malloc implementation.

To evaluate the overhead of pointer compression, the "PtrComp 32" values show the effect of transforming 32-bit pointers into 32-bit indexes (i.e. there is no compression, just overhead added). On SPARC, the added ALU overhead of pointer compression is negligible, but on AMD-64 there is a fair amount of overhead because of the extra register pressure (IA-32 has a very small integer register file). On the IBM-SP, pointer compression adds a substantial overhead to the program: the native 64-bit program is faster than the pointer compressed code until about 700 iterations in the program. On this (old) system, the memory hierarchy is fast enough, and the ALUs are slow enough that pointer compression may not make sense.

[8]Note that the IBM SP system does not support `MAP_NORESERVE`, which significantly increases the time to create a pool (thus impacting runs with a small number of iterations).

On the SPARC system, pointer compression provides a substantial speedup over PoolAlloc, and PtrComp-64 is able to *match* the performance of the 32-bit native version. On the Itanium PtrComp makes the code substantially faster across the range of iterations (but we cannot compare to a 32-bit baseline). In the case of the Opteron, PtrComp-64 is actually the fastest configuration: in 64-bit mode the Opteron can use twice as many integer registers as in its 32-bit mode, so it does not need to spill as often. On the IBM SP, performance is substantially improved with pointer compression, but can not match the 32-bit version with pool allocation because of the slow ALU. On all systems though, pointer compression improves the performance of 64-bit applications dramatically as the problem size increases. The figures also show that on all the architectures, the problem size at which performance begins to degrade rapidly is much larger for PtrComp than for PoolAlloc, showing that pointer compression significantly reduces the effective working set size of the benchmark on each of the architectures.

7.5 Related Work

If an architecture supports both 64-bit and 32-bit pointers, and if the application does not require the use of a 64-bit address space, the simplest solution is simply to compile the program in 32-bit mode, which can provide a substantial performance increase [98]. Unfortunately, this approach will not work for many applications that require 64-bit address spaces, e.g., due to genuine use of more than 4GB of memory, due to special requirements for more address space than physical memory (e.g., [121, 145, 70]), or because the system does not provide 32-bit runtime libraries (e.g. Linux IA-64). Our approach allows for selective compression of individual data structures, where each data structure is limited to 4GB of memory in the static case. In the dynamic case, there is no inherent limit.

Most recently, Adl-Tabatabai et. al. describe a trivial form of pointer compression to compile 64-bit pointers in Java programs to a 32-bit pointer model [1]. Their approach is very simple (requiring no program analysis at all), unilaterally compressing pointers to be offsets from the base of the Java memory image located in a 64-bit address space. To decompress these pointers, they add the base of the Java memory image to compressed value, allowing a Java heap size of 2^{32} bytes. This approach provides substantial performance improvements, but provides little benefit

over having the JVM produce 32-bit code directly.

Zhang and Gupta compress pairs consisting of a 32-bit integer and 32-bit pointer into two 15-bit values which are packed into a single 32-bit field [146]. They compress a pointer by replacing it with a value relative to its own address, which is effective for recursive data structures packed closely in memory. If the offset exceeds 15-bits, the pair is replaced with a pointer to an uncompressed pair on the side. They show a substantial reduction in memory consumption, cache misses, and (with custom hardware support) a reasonable performance increase on a subset of the Olden benchmarks. Unlike our work, their transformation is completely manual and only operates on pairs of values (but it can compress integers as well as pointers, and can selectively compress some fields and not others). Also, it requires specialized hardware to improve performance.

Takagi and Hiraki describe a combined hardware/software technique they dub "Field Array Compression Technique" (FACT) [131]. FACT uses manual "Instance Interleaving" [136] to split each structure definition, packing the compressed fields of multiple instances of a structure together in memory. To handle data that cannot be compressed: they always allocate enough space for both the compressed and uncompressed data. This usually improves locality though it does not reduce memory consumption. Compared with our work, FACT has higher memory consumption, requires manual transformation of the program, and requires exotic single-purpose hardware support.

An additional advantage of the macroscopic approach to pointer compression is that it allows standard compiler optimizations (e.g. loop unswitching) to statically optimize the compressed code for specific static pools. In the case of both the Zhang/Gupta and Takagi/Hiraki approaches, the compiler cannot use coarse grain optimizations because individual fields in the heap are compressed or uncompressed unrelated to each other. Using our approach, a compiler can trivially unswitch a dynamic pointer compressed loop that traverses a pool if the loop does not allocate from the pool.

7.6 Pointer Compression Summary

Transparent Pointer Compression is an aggressive technique for speculatively shrinking 64-bit pointers to 32-bit indices, without losing the generality of 64-bit pointers. We show that Pointer Compression provides both substantial performance improvements for pointer intensive codes *and* significantly reduced memory footprint for these programs.

Pointer Compression is a good demonstration of the power of macroscopic techniques. Through the use of Data Structure Analysis to identify disjoint type-homogenous data structures and Automatic Pool Allocation to partition the heap (and provide control over the memory allocation runtime), the Pointer Compression implementation is simplified to the point where it is feasible to implement. As others have found, implementing a pointer compressing technique without using macroscopic techniques requires either (extremely complex) hardware support or requires that 64-bit addressing generality be lost. Through the use of Macroscopic techniques, our approach suffers from neither of this drawbacks.

Chapter 8

Speculative Applications of Macroscopic Techniques

The primary focus of this dissertation is to demonstrate the effectiveness of macroscopic techniques for improving the memory system performance of pointer intensive programs. However, macroscopic techniques can be use for far more than just performance related applications and we have not exhausted the scope of macroscopic techniques even within this area.

This chapter briefly discusses several other techniques that are either not performance related or are not currently implemented (and are thus speculative ideas). The primary purpose of this chapter is to capture some of the ideas that we haven't investigated yet ("future work"), and describe work not performed solely by the author. These are not new research contributions (as they have not been adequately investigated), but we include them to illustrate the wide range of potential benefits and applications of this work.

These techniques make use of the four main capabilities provided by macroscopic techniques:

1. *Data structure-specific layout policies and heap segregation*: Allocating distinct instances of data structures from different pools allows compiler and run-time techniques to be customized for each instance. These techniques can use both static pool properties (e.g., type information and points-to relationships) and dynamic properties (anything recordable in per-pool metadata).

2. *Mapping of pointers to pool descriptors*: The Automatic Pool Allocation transformation provides a static many-to-one mapping of heap pointers to pool descriptors. This information is key to most transformations that exploit pool allocation because it enables the compiler to

transform pointer operations into pool-specific code sequences.

3. *Type-homogeneous pools*: Many pools are completely type-homogeneous, as shown in Section 3.4.3, even C programs. Novel compiler and run-time techniques are possible for type-homogeneous pools that would not be possible on other pools or the general heap.

4. *Knowledge of the run-time points-to graph*: One way to view pool allocation is that it partitions the heap to provide a run-time representation of the points-to graph. The compiler has full information about which pools contain pointers to other pools and, for type-homogeneous pools, where all the intra-pool and inter-pool pointers are located. Such information is useful any time pointers need to be traversed or rewritten at run-time.

The optimizations described in Chapter 6 show some simple examples of how compiler techniques can exploit these benefits, the Transparent Pointer Compression transformation (Chapter 7) makes heavy use of all of these properties, and many of the techniques described below make use of one or more of these.

8.1 Non-Performance Applications of Macroscopic Techniques

We believe that macroscopic techniques are widely applicable to areas other than performance-related compiler work. In particular, macroscopic analysis should be useful in software engineering for program understanding and visualization. It should be useful when targetting systems with partitioned memory spaces, such as deeply embedded or network processors. Finally, macroscopic techniques are also useful in the fields of distributed computing and program safety, which we discuss in more detail below.

8.1.1 Heap Safety for Languages with Explicit Deallocation

In [48]/[49], we describe an application of Automatic Pool Allocation that provides heap safety for type-safe programming languages with explicit deallocation. This work targets embedded systems that run multiple components in the same address space, e.g., a driver in a kernel, or an untrusted controller in a real-time control system [120]. Because these components are either untrusted or

potentially buggy, the runtime system needs to guarantee that one component cannot alter memory that belongs to another component.

The fundamental observation of this work is to show that if a program is otherwise type-safe (which is inferred by Data Structure Analysis), and if pointers are initialized to null when appropriate, the only way memory safety can be violated is with use-after-free (discussed here) and array bounds errors (discussed in [49]). The traditional way to solve this problem is to eliminate explicit deallocations, either by using a garbage collector or by forcing the program to use a region library which disallows all deallocations except batch disposals. Neither of these techniques is suitable for very-low-level devices: the first may introduce unpredictable pauses, slow down the program, and require increased executable sizes for GC maps, and the second requires the programmer to insert non-trivial annotations into the program.

The solution described in [48] uses pool allocation with a minor twist: explicit deallocations are preserved in the program, but the pool library is modified to never return blocks of memory back to the system, except when a pool is destroyed. Because each pool is type-homogenous, memory reuse only occurs between nodes of the same type, preventing illegal typecasts due to dangling pointers (e.g. a cast from an integer type to a pointer type). Even if a dangling pointer is dereferenced, no access to non-component state can occur.

8.1.2 Connectivity-Based Garbage Collection

Garbage Collection is a widely studied field with many implementation approaches [141, 79]. One natural application of pool allocation is to use it to either replace [135] or supplement an existing garbage collector [66, 74] for memory reclamation. Replacing garbage collection with pool allocation makes use of the "atomic destroy" property of pools, which frees memory when it is no longer reachable. Because this technique can induce unbounded space leaks [135] into the program, it is not feasible for most applications.

The most promising combination of pool allocation and garbage collection seems to be the use of partial garbage collections without write barrier overhead. In [74], Hirzel shows that lifetime and connectivity patterns are often highly correlated. Making use of this property, heap object connectivity information obtained by Data Structure Analysis, and scalar pointer information,

should allow direct implementation of CBGC. Compared to [66] and [74], using DSA as the basis for this transformation would provide a more accurate context sensitive analysis along with the ability to garbage collect from cyclic pool structures, instead of decomposing the points-to graph into a DAG of strongly connected components.

8.1.3 Data Marshalling for Pointer-Based Data Structures

Distributed computing systems that use distributed object models (e.g. CORBA and Microsoft's DCOM) are built on the idea of using data marshalling to convert complex data (e.g. structures and arrays) into a serialized format that can be transmitted over a network. The most common approach for data marshalling of recursive data structures is to marshall each node, which requires each individual node to be a distributed object.

Using macroscopic analysis and a transformation similar to static pointer compression (but which only compresses N-bit pointers to N-bit indexes), type-homogenous recursive data structures can be transformed into a "position independent" form, where indices are used to address nodes instead of pointers. In this form, code to marshall entire recursive data structures can be automatically produced by the compiler, using information from the runtime pool library to identify which nodes are allocated. This approach would reduce both the marshalling/demarshalling cost and the network bandwidth required to send a recursive data structure.

8.2 Program Performance-Related Macroscopic Applications

Program performance is the primary focus of this thesis, but we still have not been able to explore all possible applications of macroscopic techniques for improved program performance.

8.2.1 Automatic Instance Interleaving

Instance Interleaving is a technique which arranges for the fields of multiple instances of structures in a program to be interleaved with each other [136]. For example, consider a recursive data structure consisting of nodes with fields F1,F2,F3,F4. With a standard memory organization, four instances (A,B,C,D) of this node type would be laid out in memory as:

```
AF1,AF2,AF3,AF4, BF1,BF2,BF3,BF4, CF1,CF2,CF3,CF4, DF1,DF2,DF3,DF4
```

When instance interleaving is used, assuming that the fields of this structure are all the same size and that four fields fit on a cache line, memory would be organized like this instead:

```
AF1,BF1,CF1,DF1, AF2,BF2,CF2,DF2, AF3,BF3,CF3,DF3, AF4,BF4,CF4,DF4
```

The advantage of this layout is that it packs identical fields together onto a cache line. Consider a traversal of this data structure that accesses fields F1 and F2, but not F3 or F4. In the first case, each structure instance occupies an entire cache line, and traversing these four instances requires the use of four cache lines, and only half of the information each cache line is actually used. After instance interleaving, only two cache lines are accessed, reducing cache footprint of the traversal.

Instance interleaving is a powerful technique, first proposed by Truong et. al, in [136], and partially automated in [106]. They show that instance interleaving can have a large positive performance impact, but is difficult to implement. In particular, instance interleaving requires a special allocation library and requires a way to get the compiler to lay out the fields of a structure in this unusual ways. The implementation in [106] is limited in several ways: in particular, they only evaluate the transformation for very small programs, assume (but do not check) that their C programs are type-safe, performs the transformation "per type" instead of per data structure instance, does not check for memory that escapes the program, etc.

Implementing instance interleaving as a Macroscopic transformation would improve upon this in several ways, requiring implementation techniques that are very similar to the pointer compression algorithm described in Chapter 7. In particular, a macroscopic implementation could directly solve the problems with the algorithm presented in [106], making this suitable for use in a production compiler by using the following properties:

- Macroscopic analysis identifies memory that is accessed in a type-safe way.
- Macroscopic analysis identifies type-homogenous recursive data structures.
- Macroscopic analysis identifies memory objects that escape outside of the scope of analysis (e.g., those that are passed to external functions).
- Macroscopic techinques give full control over the allocation runtime library that the program allocates and frees memory with.

- Macroscopic techniques would transform each data structure instance at a time, independently of each other. This would allow different instances to have different fields collocated together with each other when profitable.

- Macroscopic techniques identify tricky cases that the algorithm must handle, such as allocation of arrays of nodes.

We believe that this aggressive application would have a large performance impact on many different programs and be reasonably straight-forward to implement.

8.2.2 Automatic use of Superpages for Improved TLB Effectiveness

TLB misses can be a significant factor that limits the performance of programs with large memory footprints. To combat this problem, architecture support for superpages has become commonplace. Superpages improve TLB "reach" by enhancing the TLB to support entries for two or more page sizes, the first is a standard size (e.g. 4K bytes) and the second is a power of two that is often much larger (e.g. 1M or 16M bytes).

Using superpages improves TLB performance by reducing the number of entries required to cover an address range. Because of this, operating system support for automatically inferring when superpages are beneficial has been investigated (e.g. [111]), focusing on how and when to promote normal pages to superpages and when to reduce them to normal pages again. However, use of superpages is not always profitable [132, 23]. In particular, superpages add increased complexity to the operating system, make swapping more expensive, and can affect working set sizes.

Macroscopic analysis and pool allocation in particular can be used to identify and increase the number of cases when superpage promotion is cost effective. In particular, a simple approach would enhance the pool runtime library (described in Section 5.1.1) to allocate superpage memory when allocating slabs that are larger than the superpage. This approach (or more aggressive ones) could increase the number of situations where use of superpages for recursive data structures is profitable, taking advantage of the data structure defragmentation properties provided by pool allocation (discussed in Section 6.3.7).

8.2.3 New Approaches for Prefetching

Prefetching for programs that use dense arrays is a well understood problem [21, 100], but prefetching for pointer-chasing traversals of recursive data structures is much harder. For example, consider Figure 8.1, a function that computes the length of a linked list.

```
struct list { int X; list *Next; };

unsigned length(list *L) {
  unsigned Length = 0;
  for (; L; L = L->Next)
    ++Length;
  return Length;
}
```

Figure 8.1: Linked-list pointer-chasing example

The problem in this case, and many other tight pointer-chasing loops, is that there is not enough work to overlap with the prefetch. Even if the prefetch for the 'next' dereference is started immediately after the previous load completes, the prefetch will not have enough time to bring the memory into cache, unless it is already there to begin with. The only general-purpose prior solution to this problem is a technique known as history-pointer prefetching [94] (also known as jump-pointer prefetching [112]).

Compressed History-Pointer Prefetching

History-pointer prefetching is one successful approach for overcoming the latency of pointer-chasing loops, which adds additional pointers to the data structure that point several nodes ahead in the traversal. Having a pointer to the node that will be needed N steps ahead in the traversal allows the prefetching code to be fetching N nodes away, which allows it to overcome almost arbitrary memory latency (assuming that these links are accurate). The primary disadvantage of history-pointer prefetching is that it simultaneously *reduces* the effectiveness of the cache by increasing the size of the list nodes. This effect is particularly bad on 64-bit systems.

Note that the inefficiency introduced by history-pointers is precisely the overhead that pointer-compression is designed to eliminate: it adds intra-data-structure pointers. For this reason, using pointer compression to compress the original and history-pointers in a data structure seems extremely powerful: it has the prefetching power of history-pointer prefetching, but without the

overhead of increasing the size of the nodes. Also, as with pointer compression, data structure analysis exposes information about when it is *safe* to modify the layout of a particular data structure, which is a prerequisite to performing automatic history-pointer prefetching for programs written in languages like C.

Pool-order prefetching

With standard heap allocation of data structure nodes, the individual nodes can be fragmented throughout memory. Automatic Pool Allocation inherently improves this situation by grouping the nodes together in memory, which has a positive effect on locality (improving effective cache line density and TLB usage). Additionally, we find that the allocation order and common traversal patterns of data structures are strongly correlated.

All of these observations lead us to believe that simple stride prefetching of data structure nodes in a pool might be an effective way to improve the performance of pointer-chasing codes. Stride prefetching is very simple and has the advantage (like history-pointer prefetching) that you can prefetch as many nodes ahead as needed to cover the latency of memory accesses. Implementing this technique and experimenting with it could provide valuable insight into the locality gains that pool allocation can provide, especially because many processors now have hardware stride prefetching hardware available.

8.2.4 Data Structure Traversal-Order Node Relocation

A common usage pattern for data structures is to have a construction/mutation phase followed by a traversal phase, followed by a destruction phase. As an example, consider a program that populates a balanced binary tree then spends a lot of time querying it. When created, the tree will require the nodes to be reordered to maintain the balancing properties, thus the common traversal orders will be unstable. However, when the program enters its query phase it will begin querying it with very similar traversal patterns.

For programs with distinct phase behavior like this, it is sometimes effective for the compiler to insert code into the program that reorders the nodes of the data structure in the expected hot traversal order. Others have observed this effect and implemented it in garbage collected systems

or with manual instrumentation, showing positive performance effects [142, 30, 29, 75].

Macroscopic techniques provide all of the information necessary to perform this transformation, automatically and safely, even for non-type-safe languages like C. This includes identification of the allocated nodes in the data structure and identification of all scalar pointers into the data structure (which would need to be updated to reorder nodes in the data structure), and information about inter-node pointers that need to be updated.

8.2.5 Identification of Coarse-Grain Parallel Work

DSA and pool allocation together provide information to make possible at least three different forms of parallelization.

Mod/Ref based region parallelization

Data Structure Analysis provides context-sensitive mod/ref information, which makes it very easy to identify function calls and other regions of code that do not interfere with each other. In short, the transformation identifies pairs of function calls whose intersected mod sets are empty, and whose mod sets do not conflict with the ref set of the other call. If these conditions are true, the calls can be executed in parallel, potentially exposing important coarse-grained parallelism, for example, parallelizing operations that occur on disjoint data structures.

We implemented a simple version of this algorithm in earlier versions of DSA, using Cilk [15] to spawn threads for the parallel calls. The primary limitation of this approach is that it does not apply to parallelism *within* a data structure, so it has limited applicability in many important situations.

Parallel processing of recursive data structures

The primary technique to expose intra-data-structure parallelism is an approach known as "Shape Analysis" [60, 117]. Shape analysis is a powerful technique that can identify a data structure as being a "list", "tree", "dag" or a general graph. One important uses of shape analysis is to parallelize computations on these data structures [71]. If each node of a data structure is recursively processed, if we know that all nodes in a data structure are processed (no early outs),

if we know the data structure isn't cyclic (i.e., a node cannot be visited more than once), and if the "processing" of each node is independent or commutative [108] it is possible to use standard "divide-and-conquer" techniques to parallelize the operation on the data structure. Unfortunately, shape analysis algorithms are also extremely expensive (often doubly exponential), limiting its use to programs that are quite small (e.g., less than a thousand lines in size) [117].

The capability that allows shape analysis to distinguish between list/tree-like data structures and DAGs is generally called the "shared" bit (either on a node [117] or a field [38] in the graph). The shared bit indicates if a memory object is pointed to by multiple heap objects. If not set, and if tree-like, the data structure may be processed with divide-and-conquer techniques. We believe that the introduction of a small amount flow-sensitivity could be added to DSA which may allow DSA to capture this property in many cases at a compile-time cost that is much less than shape analysis.

Pool-order processing of data structures

The largest gain from static analysis and pool allocation could be achieved by completely ignoring the data structure traversal pattern of the source program, eliminating pointer chasing from the program all together. Ideally, we would like to transform programs that iterate over every node in a data structure to iterate over the nodes in *pool order* instead of by traversing the pointers in the data structure. This transformation would turn sparse pointer-chasing algorithms into algorithms that are much easier to analyze (and the pool can be divided up to execute in parallel).

To safely perform this transformation, the compiler would need to identify a parallelizable data structure computation (as described in Section 8.2.5) and prove that there is only a single data structure in the pool. This goal is by far the most aggressive of any of the techniques described here. At this point it is not clear whether this goal is achievable with enough generality to make it useful in practice.

8.3 Summary

This chapter briefly described several areas for future work in the field of macroscopic data structure analysis and transformation. Several of the described techniques are extensions of other well known

approaches that are either made more powerful, more general, or automatic where the techniques were previously manual. We believe that many applications are still remaining undiscovered.

Chapter 9

Conclusion

Memory system performance is an important factor in the performance of modern systems, and is becoming even more critical over time. This thesis describes and evaluates **Macroscopic Data Structure Analysis and Optimization** – a set of aggressive, but practical, techniques that address an extremely hard problem facing compilers for modern systems: How should compilers analyze and transform programs that build and traverse recursive data structures?

Analyzing and transforming programs that use recursive data structures is an extremely difficult problem, particularly when targetting programs written in a language like C or C++. Aspects of this problem include:

- The C family of languages is very complex, supports exceptions, unpredictable setjmp/-longjmp control flow, type-unsafe pointer casts and unions, variable argument functions, etc. With the exception of garbage collection, the C language family provides a superset of the challenges faced by other languages, such as ML, Java, and Smalltalk.

- "Programs" are inherently incomplete chunks of code which are often built using external libraries, dynamically loaded libraries, etc. Compilers that assume they have knowledge of the whole program, or compilers that require changes to an ISV's build system, generally have poor acceptance for anything other than system benchmarks.

- Programs that use recursive data structures often do so with layers of helper routines (which are often recursive), use function pointers for abstraction, use `void*`'s for type genericity, etc.

- Logically distinct instances of recursive data structures are often manipulated by common routines in the program, requiring powerful analysis techniques to distinguish them.

- Compilers generally cannot reason about the dynamic layout of the heap: heap layout is controlled both by the (hard to predict) dynamic behavior of the program as well as the particular implementation of malloc/free being used, which is usually not provided by the compiler vendor. Without the ability to reason about and control layout, the compiler has limited ability to use static information infer about the program to improve program performance, and has limited ability to perform transformations that depend on heap layout.

- Modern applications are large and getting bigger: analysis and optimization time matters! Commercial compilers generally are unable to use techniques that (alone) take as much time to perform as the rest of the compile time for a program: techniques that require hours or days for programs that take minutes to compile are completely out of the question.

At root, this thesis is devoted to taking these problems and carefully breaking them down into orthogonal pieces that can be handled by purely automated techniques. With respect to the above, **The LLVM Compiler Infrastructure** (Chapter 2) is designed to canonicalize as much of the source-level complexity as possible into simple forms: it eliminates bit-fields, literal string constants, unions, complex looping structures, setjmp/longjmp, source-level exception semantics, and a tremendous amount of other source-level detail that would make interprocedural optimizations like these more complex, while preserving the important features such as the type structure, data-flow effects, data access behavior, etc. In principle, the language-independent nature of LLVM allows all of the techniques in this thesis to work unmodified for any language that targets the LLVM representation, though in practice, the techniques may have increased or reduced impact. In addition, LLVM supports aggressive and efficient link-time program analysis and optimization *without having to make significant changes to ISV makefiles.* The LLVM representation is extremely simple and light-weight, allowing interprocedural analyses and transformations to be very efficient.

Data Structure Analysis is designed to directly address the difficult program analysis problems faced by aggressive memory transformations in a consistent and unified framework. In particular, DSA computes context-sensitive points-to and mod/ref information for memory objects, information about which memory objects are accessed in a type-consistent manner, and information about which memory objects can escape the program analysis scope (thus making it illegal for a transformation to modify). This analysis is strong enough to identify distinct instances of heap

structures used by common helper functions, handles all of the difficult aspects that the LLVM IR exposes (which are inherited from C, including pointer cases, varargs, exceptions, etc), can see through `void*` casts, identifies the important structure of recursive objects, etc. Finally, despite the aggressive nature of DSA, it uses little memory and is fast and scalable in our experiments on programs spanning 4-5 orders of magnitude of code size (past 200,000 lines of code). We believe that it should continue to scale well to larger programs (as discussed in Sections 3.2.5 and 3.4.2). Finally, we demonstrate that DSA requires only a small fraction of the total compile time for the programs we tested, which we believe has never been accomplished for a general-purpose context-sensitive pointer analysis with full heap cloning.

Automatic Pool Allocation (Chapter 5) is designed to provide the compiler with the information and partial control it needs to reason about and optimize the heap layout of recursive data structures. Automatic Pool Allocation partitions the heap, changing it from a giant black box that holds memory objects into (potentially many) distinct pools in the heap which often contain a homogenous collection of memory objects with common properties. By itself this transformation often has a positive performance impact on heap-intensive programs (by increasing locality of reference among the data structure, and "deinterlacing" distinct data structures from each other), but its most important purpose is to take control of portions of the heap and enable subsequent analyses and optimizations.

Together, these techniques and algorithms are the foundation of the **Macroscopic Data Structure Analysis and Optimization** approach: we aim to identify, isolate, and optimize distinct instances of program data structures and transform them as a whole. The driving motivation for this approach is that programs are growing, frequency of code reuse and program modularization are growing, core processor speeds are growing (far outpacing the memory subsystem), and the use of recursive data structures is both prevalent and growing.

This work is primarily focused on program performance. As such, we show that Automatic Pool Allocation can have a substantial performance effect on heap intensive programs and that a number of extremely simple macroscopic techniques (Chapter 6) can be used to improve program performance even more. These simple techniques focus on directly increasing cache density by eliminating inter-object padding and memory allocator overhead, demonstrating how cooperation

between the compiler and the memory management runtime can be used to improve program performance.

As a more aggressive demonstration of the power of macroscopic techniques, Chapter 7 describes and evaluates **Transparent Pointer Compression**, which optimistically shrinks pointers in 64-bit system to 32-bit indices allowing them to grow back to 64-bits if any instance of a data structure grows over 4GB in size. We show that this technique can dramatically improve the performance of pointer-intensive programs by effectively increasing cache capacity, increasing memory bandwidth, and reducing the working set size of the program. This technique builds on Data Structure Analysis to identify type-safe data structures, identify program references to these data structures, and indicate whether a data structure ever escapes from the program. It builds on Automatic Pool Allocation to take control of the memory layout and provide runtime information and control over which nodes in a pool are dynamically allocated, allowing it to rewrite the data structure at runtime. Furthermore, it intrinsically depends on the static pointer to static pool mapping information computed by the pool allocation transformation.

In addition to these new and aggressive techniques, we show (in Chapter 4) that this framework can also be used to host analyses and transformation that use traditional alias and mod/ref analysis information, providing analysis precision that meets and exceeds other pointer analyses that require similar analysis time.

Finally, Chapter 8 describes some potential for future work in this field. In particular, though this work has primarily focused on the program performance aspects of this work, macroscopic analysis techniques are applicable to a wide variety of different program analysis and transformation problems in many domains (e.g. memory management, program safety, distributed computing, etc). We hope that continuing work in the field will expose many new ideas and approaches that we haven't even considered yet.

References

[1] Ali-Reza Adl-Tabatabai, Jay Bharadwaj, Michal Cierniak, Marsha Eng, Jesse Fang, Brian T. Lewis, Brian R. Murphy, and James M. Stichnoth. Improving 64-bit Java IPF performance by compressing heap references. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 100–110, March 2004.

[2] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. Efficient and language-independent mobile programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 127–136, 1996.

[3] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke. Llva: A low-level virtual instruction set architecture. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 205–216, San Diego, CA, Dec 2003.

[4] Alex Aiken, Manuel Fähndrich, and Ralph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 174–185, La Jolla, CA, June 1995.

[5] Wolfram Amme, Niall Dalton, Jeffery von Ronne, and Michael Franz. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2001.

[6] Lars O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[7] ANDF Consortium. The Architectural Neutral Distribution Format. `http://www.andf.org/`.

[8] Todd Austin, et al. The Pointer-intensive Benchmark Suite. `www.cs.wisc.edu/~austin/ptr-dist.html`, Sept 1995.

[9] Andrew Ayers, Stuart de Jong, John Peyton, and Richard Schooler. Scalable cross-module optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Montreal, June 1998.

[10] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, June 2000.

[11] John P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 29–41, New York, NY, USA, 1979.

[12] David A. Barrett and Ben G. Zorn. Using lifetime predictors to improve memory allocation performance. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 187–196, Albuquerque, New Mexixo, June 1993.

[13] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Seattle, Washington, November 2002.

[14] Bruno Blanchet. Escape Analysis for Java(TM): Theory and Practice. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(6):713–775, Nov 2003.

[15] Robert D. Blumofe, Christopher F. Joerg, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5^{th} ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 207–216, Santa Barbara, CA, July 1995.

[16] Greg Bollella and James Gosling. The real-time specification for Java. *IEEE Computer*, 33(6):47–54, 2000.

[17] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in real-time java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[18] Michael Burke and Linda Torczon. Interprocedural optimization: eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(3):367–399, 1993.

[19] Michael G. Burke et al. The Jalapeño Dynamic Optimizing Compiler for Java. In *Java Grande*, pages 129–141, 1999.

[20] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-conscious data placement. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 139–149, San Jose, USA, 1998.

[21] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 40–52, Santa Clara, USA, April 1991.

[22] David Chase. Implementation of exception handling. *The Journal of C Language Translation*, 5(4):229–240, June 1994.

[23] J. Bradley Chen, Anita Borg, and Norman P. Jouppi. A simulation based study of TLB performance. In *Proceedings of the International Conference on Computer Architecture (ISCA)*, pages 114–123, 1992.

[24] Juan Chen, Dinghao Wu, Andrew W. Appel, and Hai Fang. A provably sound TAL for back-end optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, Jun 2003.

[25] Ben-Chung Cheng and Wen mei Hwu. Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 57–69, Vancouver, British Columbia, Canada, June 2000.

[26] Sigmund Cherem and Radu Rugina. Region analysis and transformation for java programs. In *Proceedings of the International Symposium On Memory Management (ISMM)*, Vancouver, Canada, October 2004.

[27] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. FX!32: A profile-directed binary translator. *Proceedings of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 18(2):56–64, 1998.

[28] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 13–24, 1999.

[29] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, 1999.

[30] Trishul M. Chilimbi and James R. Larus. Using generational garbage collection to implement cache-conscious data placement. *ACM SIGPLAN Notices*, 34(3):37–48, 1999.

[31] Wei-Ngan Chin, Florin Craciun, Shengchao Qin, and Martin Rinard. Region inference for an object-oriented language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Washington, DC, June 2004.

[32] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 232–245, 1993.

[33] CodeSourcery, Compaq, et al. C++ ABI for Itanium. `http://www.codesourcery.com/cxx-abi/abi.html`, 2001.

[34] Robert S. Cohn, David W. Goodwin, and P. Geoffrey Lowney. Optimizing Alpha executables on Windows NT with Spike. *Digital Technical Journal*, 9(4), 1997.

[35] Keith Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Atlanta, GA, June 1988.

[36] Keith D. Cooper. Analyzing aliases of reference formal parameters. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 281–290, New York, NY, USA, 1985.

[37] Keith D. Cooper and John Lu. Register promotion in C programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 308–319, 1997.

[38] Francisco Corbera, Rafael Asenjo, and Emilio L. Zapata. New shape analysis techniques for automatic parallelization of c codes. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 220–227, 1999.

[39] Robert Courts. Improving locality of reference in a garbage-collecting memory management system. *Proceedings of the Communications of the ACM*, 31(9):1128–1138, 1988.

[40] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 13(4):451–490, October 1991.

[41] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 35–46, 2000.

[42] Manuvir Das, Ben Liblit, Manuel Fähndrich, and Jakob Rehof. Estimating the impact of scalable pointer analysis on optimization. In *Proceedings of the International Symposium on Static Analysis (SAS)*, pages 260–278. Springer-Verlag, 2001.

[43] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The Transmeta Code Morphing Software: Using

speculation, recovery and adaptive retranslation to address real-life challenges. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, San Francisco, CA, Mar 2003.

[44] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Snowbird, UT, June 2001.

[45] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: framework and implementations. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 261–269, 1990.

[46] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 230–241, June 1994.

[47] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 297–302, Jan 1984.

[48] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without runtime checks or garbage collection. In *Proceedings of the ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems (LCTES)*, San Diego, Jun 2003.

[49] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without garbage collection for embedded applications. *Transactions on Embedded Computing Systems*, 4(1):73–111, February 2005.

[50] Kemal Ebcioglu and Erik R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the International Conference on Computer Architecture (ISCA)*, pages 26–37, 1997.

[51] 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*

Conference on Programming Language Design and Implementation (PLDI), pages 242–256, Orlando, FL, June 1994.

[52] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 85–96, 1998.

[53] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Vancouver, Canada, June 2000.

[54] Mary F. Fernández. Simple and effective link-time optimization of Modula-3 programs. 1995.

[55] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for c. In *Proceedings of the International Symposium on Static Analysis (SAS)*, pages 175–198, London, UK, 2000. Springer-Verlag.

[56] Michael Franz and Thomas Kistler. Slim binaries. *Proceedings of the Communications of the ACM*, 40(12), 1997.

[57] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Proceedings of the Journal of the ACM*, 34(3):596–615, 1987.

[58] David Gay and Alexander Aiken. Memory management with explicit regions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 313–323, Montreal, Canada, 1998.

[59] Rakesh Ghiya and Laurie J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *International Journal of Parallel Programming*, 24(6):547–578, 1996.

[60] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 1–15, 1996.

[61] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 121–133, New York, NY, USA, 1998.

[62] Rakesh Ghiya, Daniel Lavery, and David Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2001.

[63] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2002.

[64] Dirk Grunwald and Benjamin Zorn. Customalloc: Efficient synthesized memory allocators. *SP&E*, 23(8):851–869, 1993.

[65] Brian Hackett and Radu Rugina. Region-based shape analysis with tracked locations. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 310–323, New York, NY, USA, 2005.

[66] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Berlin, Germany, June 2002.

[67] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1–12, 2000.

[68] David R. Hanson. Fast Allocation and Deallocation of Memory Based on Object Lifetimes. *Proceedings of Software–Practice and Experience*, 20(1):5–12, Jan 1990.

[69] David L. Heine and Monica S. Lam. A practical flow-sensitive and context-sensitive c and c++ memory leak detector. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 168–181, 2003.

[70] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Proceedings of Software–Practice and Experience*, 28(9):901–928, 1998.

[71] Laurie J. Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed System*, pages 35–47, 1990.

[72] Michael Hind. Which Pointer analysis Should I Use? In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2000.

[73] 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)*, pages 54–61, 2001.

[74] Martin Hirzel, Amer Diwan, and Matthew Hertz. Connectivity-based garbage collection. In *Proceedings of the ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 359–373, 2003.

[75] Xianglong Huang, Stephen Blackburn, Kathryn McKinley, Eliot Moss, Zhenlin Wang, and Perry Cheng. The garbage collection advantage: improving program locality. In *Proceedings of the ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 69–80, 2004.

[76] IBM Corp. XL FORTRAN: Eight Ways to Boost Performance. White Paper, 2000.

[77] Bertrand Jeannet, Alexey Loginov, Thomas Reps, and Mooly Sagiv. A relational approach to interprocedural shape analysis. In *Proceedings of the International Symposium on Static Analysis (SAS)*, Verona, Italy, August 2004.

[78] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, Monterey, CA, 2002.

[79] Richard Jones. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1999.

[80] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the International Conference on Computer Architecture (ISCA)*, pages 364–373, New York, NY, USA, 1990.

[81] Thomas Kistler and Michael Franz. Continuous program optimization: A case study. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(4):500–548, Jul 2003.

[82] Sumant Kowshik, Dinakar Dhurjati, and Vikram Adve. Ensuring code safety without runtime checks for real-time control systems. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, Grenoble, Oct 2002.

[83] William Landi, Barbara Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Albuquerque, NM, June 1993.

[84] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 21–34, July 1988.

[85] Chris Lattner. LLVM Alias Analysis Infrastructure. `http://llvm.cs.uiuc.edu/docs/AliasAnalysis.html`.

[86] Chris Lattner and Vikram Adve. LLVM Language Reference Manual. `http://llvm.cs.uiuc.edu/docs/LangRef.html`.

[87] Chris Lattner and Vikram Adve. Architecture for a Next-Generation GCC. In *Proceedings of the First Annual GCC Developers' Summit*, Ottawa, Canada, May 2003.

[88] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, San Jose, USA, Mar 2004.

[89] Chris Lattner and Vikram Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, Jun 2005.

[90] Chris Lattner and Vikram Adve. Transparent Pointer Compression for Linked Data Structures. In *Proceedings of the ACM Workshop on Memory System Performance*, Chicago, IL, Jun 2005.

[91] Donglin Liang and Mary Jean Harrold. Efficient points-to analysis for whole-program analysis. In *Proceedings of the European Software Engineering Conference (ESEC)*, pages 199–215, 1999.

[92] Donglin Liang and Mary Jean Harrold. Efficient computation of parameterized pointer information for interprocedural analysis. In *Proceedings of the International Symposium on Static Analysis (SAS)*, July 2001.

[93] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, 1997.

[94] Chi-Keung Luk and Todd C. Mowry. Automatic compiler-inserted prefetching for pointer-based applications. *IEEE Transactions on Computers*, 48(2):134–141, 1999.

[95] Erik Meijer and John Gough. A technical overview of the Commmon Language Infrastructure, 2002. `http://research.microsoft.com/ẽmeijer/ Papers/CLR.pdf`.

[96] Microsoft Corp. Managed extensions for C++ specification. .NET Framework Compiler and Language Reference.

[97] Ana Milanova, Atanas Rountev, and Barbara Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–11, 2002.

[98] Jeffrey C. Mogul, Joel F. Bartlett, Robert N. Mayo, and Amitabh Srivastava. Performance implications of multiple pointer sizes. In *USENIX Winter*, pages 187–200, 1995.

[99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):528–569, May 1999.

[100] Todd Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 62–73, Boston, USA, October 1992.

[101] Robert M. Muth. *Alto: A Platform for Object Code Modification.* Ph.d. Thesis, Department of Computer Science, University of Arizona, 1999.

[102] Erik M. Nystrom, Hong-Seok Kim, and Wen mei W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Proceedings of the International Symposium on Static Analysis (SAS)*, 2004.

[103] Erik M. Nystrom, Hong-Seok Kim, and Wen mei W. Hwu. Importance of heap specialization in pointer analysis. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 43–48, New York, NY, USA, 2004.

[104] David J. Pearce and Paul H. J. Kelly. A dynamic algorithm for topologically sorting directed acyclic graphs. In *Proceedings of the 3rd International Workshop on Efficient and Experimental Algorithms (WEA 2004)*, Lecture Notes in Computer Science. Springer-Verlag, 2004.

[105] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Online cycle detection and difference propagation for pointer analysis. In *Proceedings of the International IEEE Workshop on Source Code Analysis and Manipulation (SCAM)*, 2003.

[106] Rodric M. Rabbah and Krishna V. Palem. Data remapping for design space optimization of embedded memory systems. *Transactions on Embedded Computing Systems*, 2(2):186–218, 2003.

[107] Chrislain Razafimahefa. A study of side-effect analyses for java. Master's thesis, McGill University, Dec 1999.

[108] Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: a new analysis technique

for parallelizing compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):942–991, 1997.

[109] Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(2), March 1995.

[110] Ted Romer, Geoff Voelker, Denis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop*, August 1997.

[111] Theodore H. Romer, Wayne H. Ohlrich, Anna R. Karlin, and Brian N. Bershad. Reducing tlb and memory overhead using online superpage promotion. In *Proceedings of the International Conference on Computer Architecture (ISCA)*, pages 176–187, New York, NY, USA, 1995.

[112] Amir Roth and Gurindar S. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the International Conference on Computer Architecture (ISCA)*, pages 111–121, May 1999.

[113] Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 47–56, 2000.

[114] Erik Ruf. Effective synchronization removal for java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 208–218, 2000.

[115] Peter Rundberg and Fredrik Warg. The FreeBench v1.0 Benchmark Suite. `http://www.freebench.org`, Jan 2002.

[116] Barbara Ryder, William Landi, Philip Stocks, Sean Zhang, and Rita Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(2):105–186, March 2001.

[117] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(1), January 1998.

[118] Robert Sedgewick. *Algorithms*. Addison-Wesley, Inc., Reading, MA, 1988.

[119] Matthew L. Seidl and Benjamin G. Zorn. Segregating heap objects by reference behavior and lifetime. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 12–23, San Jose, USA, 1998.

[120] Lui Sha. Dependable system upgrades. In *Proceedings of IEEE Real Time System Symposium*, 1998.

[121] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings ACM Conference on Computer and Communications Security (CCS '04)*, pages 298–307, 2004.

[122] Ran Shaham, Eran Yahav, Elliot K. Kolodner, and Mooly Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. In *Proceedings of the International Symposium on Static Analysis (SAS)*, San Diego, USA, June 2003.

[123] Zhong Shao, Christopher League, and Stefan Monnier. Implementing Typed Intermediate Languages. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 313–323, 1998.

[124] Anand Shukla. Lightweight, cross-procedure tracing for runtime optimization. Master's thesis, Computer Science Deptartment, University of Illinois at Urbana-Champaign, Urbana, IL, Aug 2003.

[125] James E. Smith, Timothy Heil, Subramanya Sastry, and Todd Bezenek. Achieving high performance via co-designed virtual machines. In *Proceedings of the International Workshop on Innovative Architecture (IWIA)*, 1999.

[126] Amitabh Srivastava and David Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, Dec. 1992.

[127] T.B. Steel. Uncol: The myth and the fact. *Annual Review in Automated Programming 2*, 1961.

[128] Bjarne Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *Proceedings of the International Conference on Compiler Construction (CC)*, pages 136–150, London, UK, 1996.

[129] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 32–41, Jan 1996.

[130] Phil Stocks, Barbara G. Ryder, William Landi, and Sean Zhang. Comparing flow and context sensitivity on the modification-side-effects problem. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 21–31, 1998.

[131] Masamichi Takagi and Kei Hiraki. Field array compression in data caches for dynamically allocated recursive data structure. In *Proceedings of 5th International Symposium on High Performance Computing (ISHPC'03)*, pages 127–145, October 2003.

[132] Madhusudhan Talluri, Shing I. Kong, Mark D. Hill, and David A. Patterson. Tradeoffs in supporting two page sizes. In *Proceedings of the International Conference on Computer Architecture (ISCA)*, pages 415–424, 1992.

[133] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):724–768, July 1998.

[134] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 188–201, 1994.

[135] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, pages 132(2):109–176, February 1997.

[136] Dan N. Truong, François Bodin, and André Seznec. Improving cache behavior of dynamically allocated data structures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 322–329, October 1998.

[137] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings of the ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1987.

[138] Frdric Vivien and Martin Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 35–46, 2001.

[139] David Wall. Global register allocation at link-time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Palo Alto, CA, 1986.

[140] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 131–144, 2004.

[141] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, number 637, Saint-Malo (France), 1992. Springer-Verlag.

[142] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 177–191, 1991.

[143] Robert P. Wilson and Monica S. Lam. Effective context sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, June 1995.

[144] Eran Yahav and G. Ramalingam. Verifying safety properties using separation and hetero-

geneous abstractions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 25–34, New York, NY, USA, 2004.

[145] Curtis Yarvin, Richard Bukowski, and Thomas Anderson. Anonymous RPC: Low-latency protection in a 64-bit address space. In *USENIX Summer*, pages 175–186, 1993.

[146] Youtao Zhang and Rajiv Gupta. Data compression transformations for dynamically allocated data structures. In *Proceedings of the International Conference on Compiler Construction (CC)*, Apr 2002.

[147] Craig B. Zilles. Benchmark health considered harmful. *ACM SIGARCH Computer Architecture News*, 29(3):4–5, 2001.

Author's Biography

Chris Lattner attended elementary school in Beaverton Oregon (suburbia), moving to Banks Oregon (the country) for middle school and high school. After high school, he attended the University of Portland in Portland, Oregon (the city), and was awarded a Bachelors degree in Computer Science. For graduate school, he moved to Urbana, IL (the corn fields) and attended the University of Illinois at Urbana Champaign, earning his M.S. and Ph.D. degrees. Post graduation, he is joining Apple Computer in Cupertino, California (the silicon valley), where he will continue development of LLVM and find new cool ways for it to improve their products. Among other things, Chris thinks that author biographies should not be compulsory for Ph.D. dissertations at the University of Illinois. For the curious, Chris's work history, publications and many other things are available on his web page (`http://nondot.org/sabre/`).