© Copyright by Chris Arthur Lattner, 2002

LLVM: AN INFRASTRUCTURE FOR MULTI-STAGE OPTIMIZATION

BY

CHRIS ARTHUR LATTNER

B.S., University of Portland, 2000

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2002

Urbana, Illinois

Abstract

Modern programming languages and software engineering principles are causing increasing problems for compiler systems. Traditional approaches, which use a simple compile-link-execute model, are unable to provide adequate application performance under the demands of the new conditions. Traditional approaches to interprocedural and profile-driven compilation can provide the application performance needed, but require infeasible amounts of compilation time to build the application.

This thesis presents LLVM, a design and implementation of a compiler infrastructure which supports a unique *multi-stage* optimization system. This system is designed to support extensive interprocedural and profile-driven optimizations, while being efficient enough for use in commercial compiler systems.

The LLVM virtual instruction set is the glue that holds the system together. It is a low-level representation, but with *high-level type information*. This provides the benefits of a low-level representation (compact representation, wide variety of available transformations, etc.) as well as providing high-level information to support aggressive interprocedural optimizations at link- and post-link time. In particular, this system is designed to support optimization in the field, both at run-time and during otherwise unused idle time on the machine.

This thesis also describes an implementation of this compiler design, the LLVM compiler infrastructure, proving that the design is feasible. The LLVM compiler infrastructure is a maturing and efficient system, which we show is a good host for a variety of research. More information about LLVM can be found on its web site at: `http://llvm.cs.uiuc.edu/`

Acknowledgments

This thesis would not be possible without the support of a large number of people who have helped me both in big ways and little.

In particular, I would like to thank my advisor, Vikram Adve, for his support, patience, and especially his trust and respect. He has shown me how to communicate ideas more effectively and how to find important and meaningful topics for research. By being demanding, understanding, and allowing me the freedom to explore my interests, he has driven me to succeed.

The inspiration for this work certainly stems from one person: Tanya. She has been a continuous source of support, ideas, encouragement, and understanding. Despite my many late nights, unimaginable amounts of stress, and a truly odd sense of humor, she has not just tolerated me, but loved me.

Another person who made this possible, perhaps without truly understanding his contribution, has been Brian Ensink. Brian has been an invaluable sounding board for ideas, a welcoming ear to occasional frustrations, provider of mints, and helped me tremendously with the paperwork and other annoying requirements of submitting a thesis.

There are two others who have been very important to me during my time here in Champaign-Urbana. Jim Oly has, on numerous occasions, provided a critical eye for my writing, and more importantly has been a great friend. Finally, this work would not be possible without the Coca-Cola Bottling Company, which provided the raw materials that fueled my work, sometimes late into the night.

Table of Contents

List of Figures

1 Introduction

Modern programming languages and software practices aim to support more reliable, modular, and dynamic software applications – increasing programmer productivity, and providing higher-level semantic information to the compiler. However, in many cases, these features impose a run-time performance penalty on compiled applications.

At the same time, microprocessors are continuing to evolve at a break-neck pace. Pipelines get deeper, caches are gaining additional levels, and memory access times are getting slower (relative to the CPU). To compensate, hardware designers are exposing more parallel execution resources and integrating features that were once the sole domain of compiler developers: register-renaming engines and reordering buffers, for example.

Situated between the modern programming language and the architecture, the compiler is responsible for making the application perform as well as possible. Compilers do this by eliminating provably unnecessary overhead from the program and by making effective use of the resources exposed by the processor. The solution to both problems above is conceptually simple: increase the scope of analysis and optimization, allowing the compiler to do a better job. Unfortunately, traditional approaches to compilation are poorly suited to handling these new demands placed upon the compiler.

This thesis describes the **Low-Level Virtual Machine** (LLVM), a compiler infrastructure which is well-suited for modern programming languages and architectures. LLVM is designed to achieve three critical goals:

1. Enable an aggressive *multi-stage* optimization strategy, providing maximum performance.

2. Serve as the host for leading edge research and development, providing a strong foundation for both current and future projects.

3. Operate transparently to the end-user (a developer), behaving identically to a standard system compiler (including realistic compilation times).

LLVM is designed to address one simple observation: human patience is limited. Supporting this, LLVM provides excellent end-user application performance, good compile-time performance for application developers, and a productive research environment for compiler developers. In order to understand the design decisions made for LLVM, we first describe existing approaches to dealing with these issues as well as the deficiencies of these approaches.

1.1 Existing Compiler System Approaches

Compiler systems have enjoyed a broad range of research into different methods of producing high-performance executables. Most aggressive compilers use one or more of the following techniques, often very effectively. Unfortunately, these techniques are not suitable for a compiler which must produce high-performance executables *and* achieve low compilation times. The three techniques are link-time interprocedural optimization, run-time dynamic optimization, and profile-driven optimization.

1.1.1 Traditional Approaches to Link-Time Interprocedural Optimization

Interprocedural (or whole-program) optimization is a highly effective technique for providing high performance executables. The underlying idea is to gather as much of the program together into one place as possible, increasing the scope of analysis and transformation beyond a single translation unit. The most important decision determining the scope of optimizations possible in an interprocedural optimizer is the following: at which "level" will the program be represented?

In existing interprocedural optimizers, there are two answers to this question:

1. **Very Low Level** - Machine Code

   A large amount of research has gone into performing interprocedural optimizations on machine code [5, 16, 24, 34, 40, 11, 35] at either link-time or run-time. The advantage of these systems is that they often work with unmodified front-end compiler systems, allowing the developer to use any compiler they want.

   These systems suffer from a very important limitation: machine code simply does not provide enough high-level information to support aggressive interprocedural analyses or transformations. These systems typically target very low-level transformations such as interprocedural register allocation, inlining, and trace construction.

2. **Very High Level** - Abstract Syntax Trees (AST)

   In order to solve this problem, compiler developers invented techniques to preserve source-level information all the way until link-time. In most cases [19, 4, 13, 37, 46], this is implemented by writing out the high-level compiler Intermediate Representation (IR) to disk at compile time. At link time, the linker reads these serialized versions of the program AST, combining them, optimizing them, and finally performing all code generation at link time.

   This approach solves the problem with the very-low-level approach to interprocedural optimization (lack of high-level information), but at a high cost. Because almost all compilation is postponed to link time, any change to a single source file requires almost complete recompilation of the program. Additionally, compiler IRs are often proprietary, severely restricting the amount of interoperability between compilers. In fact, in some cases, even different versions of the same compiler cannot communicate because they are dependent on the memory layout of the intermediate representation.

1.1.2 Traditional Approaches to Run-Time Optimization

Like interprocedural optimization, there are multiple ways of approaching run-time optimization. The most common approach is to simply ignore dynamic optimization completely, as most static compilers do. By not using any run-time optimization or monitoring, however, the entire range of dynamic program behavior is missed. For this reason and others, run-time optimization has become a frequently applied technique for achieving high performance in modern systems.

The two most common types of dynamic optimization systems are as follows:

1. **High Level Language Virtual Machines**

   Run-time optimization and Just-In-Time (JIT) compilation are very common among the class of high-level language Virtual Machines (VMs). These VMs often target very dynamic languages, such as SmallTalk [21], Self [44], Java [22], and C# [32], and use a machine-independent byte-code input which encodes these languages at a very high-level (effectively at the AST level). By using a virtual machine and a very-high-level input program representation, these systems are able to provide platform portability and security services in addition to reasonable performance.

Unfortunately, a high-level representation presents the same problem to a run-time optimizer that it does for an interprocedural link-time optimizer: no substantial optimizations can be performed at compile time. In the case of a JIT compiler, this means that the dynamic compiler must spend valuable run-time cycles performing mundane optimizations like common copy propagation, which reduces the number of run-time cycles available for more interesting optimizations. On the other hand, these high-level representations do provide the dynamic compiler with a rich information source, allowing a wide variety of interesting optimizations if run-time cycles can be spared.

2. **Architecture Level Virtual Machines** and **Dynamic Translators**

   At the other end of the spectrum are machine code reoptimizers and instruction set translators. These systems either manipulate native machine code to achieve higher performance [5] or dynamically translate machine code between different architectures [16, 24]. These systems have drawbacks and applications similar to the machine code interprocedural optimizers. They tend to work very well with trace formation and optimization, which rely on highly accurate profiling information, but are incapable of high-level restructuring transformations.

1.1.3 Traditional Approaches to Compile-time Profile-Driven Optimization

Profile-driven optimization [23] is an important technique which uses the estimated run-time behavior of the application to improve its performance (often by optimizing common cases at the expense of uncommon cases). The traditional way to integrate profiling into a compilation system is to split the standard compile and link stages of compilation into a five stage process.

The first stage of compilation compiles the program, but inserts profiling instrumentation into the program to cause it to gather some form of profile information at run-time. The second stage links these instrumented object files into an instrumented executable. The third stage of profile-driven optimization requires the developer of the application to run the generated executable through a series of test runs, which are used to generate the profile information for the application.

Finally, the fourth and fifth stages recompile the program (often from source) and relink it, using the collected profile information to optimize the program.

While profile-driven optimization is an important tool that can have a large impact on the final execution performance of the application, this approach has many suboptimal features. First, profile information is only useful if it is accurate [39, 11]. Realistic programs (as opposed to benchmarks) often have many different ways to use the application, and the usage pattern of the developer's profile runs may not match the usage pattern of a particular user. Because of this, static profile information may actually be counter-productive: optimizations based on it may slow down the cases the user is actually encountering.

The larger problem, however, is that developers are often not even willing to use profile guided optimization at all because it is too cumbersome [11]. In order to use this technique, developers must modify build processes and testing cycles to account for the new five step process. If the application is not easily scriptable (because it is graphical, for example), manually exercising the program to build profile information is even more error prone and expensive.

1.2 Multi-stage Optimization with LLVM

The LLVM system architecture (described in Chapter 2) is designed to address these problems in traditional systems. Briefly, the static compilers in the LLVM system compile source code down to a low-level representation that includes high-level type information: The LLVM Virtual Instruction Set (described in Chapter 3). This allows the static compiler to perform substantial optimizations at compile time, while still communicating high-level information to the linker.

At link time, the program is combined into a single unit of LLVM virtual instruction set code, and is interprocedurally optimized (several examples of high-level interprocedural optimizations are included in Chapter 4). Once the program has been completely optimized, machine code is generated and a native executable is produced. This executable is native machine code, but it also includes a copy of the program's LLVM bytecode for later stages of optimization.

The LLVM run-time optimizer simply monitors the execution of the running program, gathering profile information. When the run-time optimizer determines that it can improve the performance of the program through a transformation, it may do so through two routes: either direct modification of the already optimized machine code or new code generation from the attached LLVM bytecode. In either case, the LLVM bytecode provides important high-level control flow, data flow, and type information that is useful for aggressive run-time optimizations.

Some transformations are too expensive to perform at run-time, even given an efficient representation to work with. For these transformations, the run-time optimizer gathers profile information, serializing it to disk. When idle time is detected on the user's computer, an offline reoptimizer is used to perform the most aggressive profile-driven optimizations to the application. The offline optimizer is equivalent in power to the link-time optimizer. The difference is that the offline optimizer uses profile *and* interprocedural analysis information to improve the program, where the link-time optimizer must do without profile information.

Note that this system collects profile information **in the field**, which provides the most accurate information possible, and does not interfere with the development process at all. The use of the LLVM virtual instruction set allows work to be offloaded from link-time to compile-time, speeding up incremental recompilations. Also, because all of the components operate on the same representation, they can share implementations of transformations.

1.3 Research Contributions of this Thesis

The first main contribution of this thesis is to show that *aggressive interprocedural analyses and transformations* may be performed on a *low-level representation*, provided that it has *high-level type information.*

The second main contribution of this thesis is to show how the low-level representation can be used to build a compiler system which includes sophisticated optimizations, while still being practical to use. Specifically, this system:

- ... fits into the standard build model, operating as a drop-in replacement for pre-existing tools.
- ... supports sophisticated interprocedural analyses and transformations at both post-link and link-time (with and without profile information, respectively).

- ... enables novel strategies for run-time optimization which can operate on machine code with the advantage of having high-level information available.
- ... collects profile information and reoptimizes the program *in the field*, allowing for the most accurate profile information and, consequently, the highest performance applications.

The third main contribution of this thesis is an implementation of this design. This implementation, the LLVM Compiler Infrastructure, is a solid infrastructure which is hosting a variety of current research (see Chapter 4). Of particular note is the fact that LLVM has successfully been used as the host infrastructure for an advanced compilers class (Section 4.4.3). Students tend to be much less forgiving than researchers about a poor design, lack of documentation, buggy implementation, or poor extensibility, so this demonstrates a great deal of maturity. In time, we hope to be able to make LLVM available to researchers outside of the University of Illinois.

1.4 Organization of this Thesis

In order to fully understand the design of the LLVM System Architecture, Chapter 2 discusses the tools and how they work together. Chapter 3 describes the features of the LLVM virtual instruction set which make it suitable for use as the common program representation in the LLVM system. Chapter 4 describes some applications of the LLVM system, showing that a low-level representation can successfully host a variety of aggressive analyses and transformations, if accompanied with type information. Chapter 5 evaluates the LLVM compiler infrastructure in terms of maturity, productivity, and performance. Chapter 6 briefly describes related work in the field, and Chapter 7 concludes the work.

2 LLVM System Architecture

The LLVM system is designed around a multi-stage approach to compilation, as briefly described in Section 1.2. This chapter continues the discussion, describing the individual components of the LLVM system design and their interfaces. This compilation strategy is unique in the fact that it allows aggressive optimization throughout the lifetime of the application while remaining practical.

2.1 The High-Level Design of an LLVM Based Compiler

Compared with current compilation systems, the LLVM system is designed to perform more sophisticated transformations at link-time, run-time, and after the software is installed in the field. In order to be realistically deployable, however, the LLVM compiler must integrate well with existing build schemes, and it must be efficient enough to be used in common scenarios. This section describes the overall approach to compilation, explaining how these requirements are addressed. A diagram of the overall LLVM system is shown in Figure 2.1.

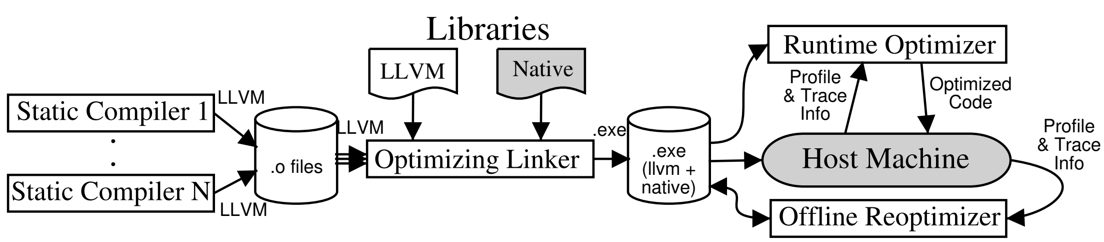


Figure 2.1: LLVM system architecture diagram

Traditional compilers break the compilation process into two steps: compile and link. Separating the two phases provides the benefits of separate compilation: only the translation units modified need to be recompiled (although the entire application must still be relinked). A traditional compiler compiles source code to an object file (`.o`) containing machine code, and the linker combines these object files together with libraries to form an executable program. In a simple system, the linker typically does little more than concatenate the object files and resolve symbol references.

The LLVM approach retains the distinction between compile and link time, allowing it to retain the advantages of separate compilation. Instead of compiling directly to machine code, however, the static compiler front-ends (described in Section 2.2) emit code in the LLVM virtual instruction set. The LLVM optimizing linker (described in Section 2.3) combines these LLVM object files, optimizes them, and finally integrates them into a native executable which it writes to disk. This organization permits sophisticated interprocedural optimizations to be performed at link time (where they are most effective).

The executable written by the optimizing linker contains native machine code directly executable on the host architecture as well as a copy of the LLVM bytecode for the application itself[1]. As the application is executed in the field, a runtime reoptimizer may monitor the execution of the program, collecting profile information about typical usage patterns for the application.

Optimization opportunities detected from application behavior may cause the runtime reoptimizer to dynamically recompile and reoptimize portions of the application (using the stored LLVM bytecode). However, some transformations may be too expensive to perform directly at runtime. For these transformations, idle time on the machine is used by an offline optimizer to recompile the application using aggressive interprocedural techniques and the accurate profile information detected from the end-user's actual usage patterns.

The key points of the high-level LLVM system design is that the LLVM virtual instruction set (described in more detail in Chapter 3) is used the communicate between the different tools, and the tools fit into a standard development framework. Operating on a common representation allows the transformations to be shared between the different components of the system. Specific aspects of each component are described below.

2.2 Compile Time: Front-end & Static Optimizer

The LLVM system is designed to support multiple language front-ends, each of which translates the supported source languages into the LLVM virtual instruction set. Each static compiler performs as much optimization as possible on each translation unit to reduce the amount of work required of the link-time optimizer.

[1]LLVM bytecode is contained in a special section of the executable, so it is only paged into memory when and if accessed by the runtime optimizer.

The primary job of the language front-end is to translate from the source language to the LLVM virtual instruction set, but it can also perform language-specific optimizations as well. For example, a C or C++ front-end can optimize the call "`printf("hello\n");`" into "`puts("hello");`", because the high-level semantics of the functions in question are defined by the C standard.

Because all of the LLVM transformations are modular and shared, static compilers can choose to use some (or all) of the LLVM infrastructure transformations to improve their code generation capabilities. Note that this includes the interprocedural optimizations used by the link-time optimizer, which may be used on the more limited scope of a translation unit as well as the larger scope at link-time.

Key to the design of the LLVM virtual instruction set is the ability to support arbitrary source languages through a common low-level type system. Unlike high-level virtual machines, the LLVM type system does not specify an object model, memory management system, or specific exception semantics that each language must use. Instead, LLVM only directly supports the lowest-level type constructors, such as pointers, structures, and arrays, relying on the source language to map the high-level type system to the low-level one. In this way, LLVM is language independent in the same way a microprocessor is: all high-level features are mapped down to simpler constructs.

2.3 Link Time: Linker & Interprocedural Optimizer

Link time is the first phase of the compilation process where the majority[2] of the program is available for analysis and transformation. As such, the LLVM optimizing linker is a natural place to perform aggressive interprocedural optimizations across the entire program.

All transformations are modular in LLVM, allowing the LLVM optimizing linker to use traditional scalar optimizations (employed by the static compilers), to clean up the results of large scale interprocedural optimizations. Like the static compilers, the link-time optimizations operate on LLVM bytecode directly; thus they are able to take advantage of the high-level information encoded into them, making them more effective. For example, the Automatic Pool Allocation transformation, described in Section 4.3, fundamentally requires the type information provided by LLVM, while the Data Structure Analysis transformation, described in Section 4.2, is made more

[2]Note that shared libraries and system libraries may not be available for analysis at link time.

accurate due to the SSA form used by the LLVM virtual instruction set.

The design of the compile-time and link-time optimizers permits the application 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[3]. The link-time interprocedural optimizer can then process these interprocedural summaries as input instead of having to compute results from scratch. This technique reduces the amount of analysis that must be performed when only a few translation units need to be recompiled, thus potentially saving a substantial amount of compile time [7].

Once link-time optimization has been completed, a code generator appropriate to the target is selected to translate from LLVM to native code for the current platform. If the user decides to use the post-link optimizers, a copy of the compressed LLVM bytecode is included into the executable itself. Including the bytecode directly in the generated executable eliminates the possibility that the runtime or offline optimizers will acquire the wrong bytecode for a given program.

2.4 Run Time: Profiling & Reoptimization

One of the key research goals of the LLVM project is to develop a new strategy for runtime optimization. This strategy is built around the model of gathering profile information at runtime, and using it to control reoptimization and recompilation of the program from the LLVM bytecode.

2.4.1 Gathering Profile Information at Run Time

Avoiding the traditional approach to profile guided optimization (described in Section 1.1.3) is an important goal of the LLVM system. There are two primary disadvantages to the traditional approach: the profile information measures the usage pattern of the developer (not the user), and developers rarely actually use profile guided feedback. The use of runtime profiling (in the field) eliminates these two problems: the end-user of the application provides profile runs as they use the application, with no extra work by the developers.

Note that the runtime reoptimizer may use a variety of different techniques to collect profile

[3]Note that this is achieved without building a program database or deferring the compilation of the input source code until link-time. This eliminates the possibility that the program database could be out of sync with the object files.

information, ranging from PC sampling techniques [2] (to find hot functions and loops) to path profiling [6] (to determine the hot paths through a complex region of code). Over the lifetime of the application, the runtime optimizer will eventually become dormant, only changing the program if strong phase behaviors occur which can benefit from continuous optimizations.

2.4.2 The LLVM Approach to Run Time Optimization

Unlike other virtual machine based systems, the LLVM runtime optimizer can choose to do lightweight optimizations directly on the precompiled native machine code while referring to the LLVM bytecode for high-level information about dataflow and types. This capability is enabled through the use of detailed mapping information, which maps between the native and LLVM code representations for the program.

This information allows for simple transformations (code layout, for example) to be implemented safely (due to control flow information from the LLVM code) and efficiently (because the machine code is already generated for the code). More aggressive transformations (based on the results of value profiling [8], for example) may instead elect to modify the LLVM bytecode for the program and regenerate machine code from it instead. This approach is useful optimizations of medium complexity. For very expensive optimizations, the offline reoptimizer is used.

2.5 Idle Time: Offline Reoptimizer

Some types of applications are not particularly amenable to runtime optimization: these applications often have a large amount of code, none of which is very "hot". Because of this, the runtime optimizer cannot afford to spend a significant amount of time improving any one piece of the code, although it can probably still detect the most frequent paths executed by the program.

In order to support these types of applications and to support other optimizations which require potentially expensive analyses, the offline reoptimizer is available. It is designed to be run during idle time on the user's computer, allowing it to be much more aggressive than the runtime optimizer.

The offline reoptimizer combines profile information gathered by the runtime optimizer with the LLVM bytecode to reoptimize and recompile the application. In this way it is able to perform aggressive profile driven interprocedural optimization without competing with the application for processor cycles. As the usage pattern of the application changes over time, the runtime and offline reoptimizers coordinate to ensure the application is performing at its peak capability.

3 LLVM Virtual Instruction Set

The LLVM system architecture is designed to produce the highest performance executables through an aggressive system of continuous optimization. One of the key factors that differentiates LLVM from other systems, however, is the program representation it uses. This program representation must be low-level enough to allow significant amounts of optimization in the early phases of compilation, while being high-level enough to support aggressive link- and post-link time optimizations.

The LLVM virtual instruction set is designed as a *low-level* representation with *high-level* type information. It provides extensive language independent type information about all values in the program, exposes memory allocation directly to the compiler, and is specifically designed to have uniform abstractions. This chapter discusses the major features of the LLVM virtual instruction set. The syntax and semantics of each instruction are defined in the LLVM reference manual [29].

3.1 Overview of the LLVM Virtual Instruction Set

The LLVM instruction set represents a virtual architecture that captures the key operations of ordinary processors but avoids machine specific constraints such as physical registers, pipelines, low-level calling conventions, or traps. LLVM provides an infinite set of typed virtual registers which can hold values of primitive types (integral, floating point, or pointer values). The virtual registers are in Static Single Assignment (SSA) form [15], a widely used representation for compiler optimization, as explained in Section 3.2.1. The LLVM type system is explained in more detail in Section 3.3. The LLVM virtual instruction set also has a unique mechanism for explicit representation of exceptional control flow, which is described in Section 3.5.

LLVM programs transfer values between virtual registers and memory solely via `load` and `store` operations using typed pointers. Memory is partitioned into a global area, stack, and heap (with procedures being treated as global objects). Objects on the stack and heap are allocated using `alloca` and `malloc` instructions respectively, and are accessed through the pointer values returned by these operations. Stack objects are allocated in the stack frame of the current function and are automatically freed when control leaves the function. Heap objects must be explicitly freed using a `free` instruction. The motivation and implementation of these operations are explained in Section 3.4 below.

Note that LLVM is a virtual instruction set: it does not define runtime and operating system functions such as I/O, memory management (in particular, garbage collection), signals, and many others. These features are defined by runtime libraries and APIs that programs link against. On the other hand, the LLVM virtual instruction set is a *first class language* which has a textual, binary, and in-memory representation. The implications of this decision are discussed in Section 3.6.

3.2 Three-Address Code

Three-address code has been the representation of choice for RISC architectures and language-independent compiler optimizations for many years. It is very close in spirit to machine code, with a small number of simple, orthogonal operations. Three-address code can be easily compressed, allowing for high density LLVM files.

Most LLVM operations, including all arithmetic and logical operations, are in three-address form: they take one or two operands and produce a single result. LLVM includes a standard and very orthogonal set[1] of arithmetic and logical operations: `add`, `sub`, `mul`, `div`, `rem`, `not`, `and`, `or`, `xor`, `shl`, `shr`, and `set`*`cc`*. The latter (`set`*`cc`*) is actually a collection of comparison instructions with different operators (e.g., `seteq`, `setlt`, etc...) that produce a boolean result. In addition to simple binary instructions, some instructions take 0, 3, or a variable number of operands. Important examples include `call` instructions and the `phi` instruction used to represent code in SSA form.

A key point is that LLVM instructions are polymorphic: a single instruction (like `add`) can operate on several different types of operands. This greatly reduces the number of distinct opcodes. In particular, we do not require different opcodes for operations on signed and unsigned integers, single- or double-precision floating point values, arithmetic or logical shifts, etc. The types of the operands automatically define the semantics of the operation and the type of the result, and they must follow strict type rules defined in the reference manual [29].

For example, Figure 3.1 illustrates some simple LLVM operations. In this example, the type

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

```
  %X = div int 4, 9                  ; Signed integer division
  %Y = div uint 12, 4                ; Unsigned integer division
  %cond = setlt int %X, 8            ; Produces a boolean value
  br bool %cond, label %True, label %False
True:
  ...
```

Figure 3.1: LLVM code snippet illustrating typed operations

information determines whether to perform a signed or unsigned division, and whether the comparison should be signed or unsigned.

3.2.1 Static Single Assignment Form

LLVM uses Static Single Assignment (SSA) form as its primary code representation. A program is said to be in SSA form if each of its variables is defined exactly once, and each use of a variable is dominated by that variable's definition. SSA form greatly simplifies many dataflow optimizations because only a single definition can reach a particular use of a value, and finding that definition is trivial. It also enables fast flow-insensitive algorithms to achieve many of the benefits of flow-sensitive algorithms without expensive dataflow analysis (sometimes referred to as the sparseness property).

One implication of the "single definition" property is that each instruction that computes a value (e.g., `add int %x, %y`) implicitly creates a new virtual register holding that value. The value may be given an explicit name (e.g., `%z = add ...`); if not, a unique name is automatically assigned by the LLVM system. This property enables "uses" in LLVM to refer directly to the operation that computes the value, enabling efficient traversal of def-use information.

When control flow is taken into account, simple variable renaming is not enough for code to be in valid SSA form. To handle control flow merges, SSA form defines the ϕ function, which is used to select an incoming value depending on which basic block control flow came from. LLVM provides a `phi` instruction which corresponds to the SSA ϕ-node. The syntax of this instruction is:

```
<result> = phi <type> [<val0>, <label0>], ...  , [<valN>, <labelN>]
```

`result` is assigned the value `val0` if control reaches this instruction from the basic block labelled `label0`, `val1` if control reaches here from basic block `label1`, and so on. All the `phi` instructions in a basic block must appear at the beginning of the basic block. Figure 3.2 shows an example function which requires ϕ nodes.

```
int pow(int M, unsigned N) {
  unsigned i; int Result = 1;
  for (i = 0; i != N; ++i)
    Result *= M;
  return Result;
}
```

Figure 3.2: C code for example loop body

Figure 3.3 shows the LLVM code representing the loop body.

```
Loop:
  %result  = phi  int [ 1, %LoopHeader ], [ %result2, %Loop ]
  %i       = phi uint [ 0, %LoopHeader ], [ %i2,      %Loop ]
  %result2 = mul int %result, %M              ; Result *= M
  %i2      = add uint %i, 1                   ; i = i + 1
  %cond    = setne uint %i2, %N               ; i != N
  br bool %cond, label %Loop, label %Exit
```

Figure 3.3: LLVM code for example loop body

As noted before, the virtual registers in LLVM are in SSA form while values in memory are not. This dramatically simplifies transformations because scalars cannot have aliases.

3.3 High-Level Type Information

LLVM is a strictly typed representation, where every SSA value and memory location has an associated type, and all operations obey strict type rules. This type information enables a broad class of *high-level* transformations on *low-level* code. In addition, type mismatches can be used to detect errors in optimizations by the LLVM consistency checker.

The LLVM type system includes source language independent primitive types (void, bool, signed and unsigned integers from 8 to 64 bits, floating-point values in single and double precision, and opaque) and constructive types (pointers, arrays, structures, and functions). These types are language-independent data representations that are mapped from higher-level language types. For example, classes in C++ with inheritance and virtual methods can be represented using structures for the data values and a typed function table with indirect function calls for inheritance. This permits many high-level language-independent optimizations (e.g., virtual function resolution) to be performed on the LLVM code. Some examples illustrating LLVM types are shown in Figure 3.4.

```
%arrayty = [2 x [3 x [4 x uint]]]  ; 2x3x4 array of unsigned integer values

%aptr = [4 x int]*                 ; Pointer to array of four int values

%funcptr = float (int, int *) *    ; Pointer to a function that takes an int
                                   ; and a pointer to int, returning float

%strty = { float, %funcptr }       ; A structure, where the first element is a
                                   ; float and the second element is the
                                   ; %funcptr pointer to function type defined
                                   ; previously
```

Figure 3.4: Examples of LLVM types

All LLVM instructions are strictly typed, and all have restrictions on their operands to simplify transformations and preserve type correctness[2]. For example, the `add` instruction requires that both operands are of the same type, which must be an arithmetic (i.e., integral or floating-point) type, and it produces a value of that type. The `load` instruction requires a pointer operand to load from. The `store` instruction requires a value of some type (say, τ) to store and a pointer to store into, which must be a pointer to that type (τ*). Some examples of malformed code are shown in Figure 3.5.

```
uint %testfunc() {
  %v = load int 4               ; Must load through a pointer
  store int 42, float* %fptr    ; Cannot store 'int' through 'float*'
  %Val = add int %Val, 0        ; Definition does not dominate use
  ret int* null                 ; Cannot return 'int*' from fn returning 'uint'
}
```

Figure 3.5: Examples of malformed LLVM code

[2]These restrictions make the LLVM code more compact, as explained in Section 3.2

Type information enables high-level information to be easily extracted from the low-level code, enabling novel transformations at link time. To do this, however, it must be possible to allow type-safe access to fields of data in memory. For this reason, a critical instruction in LLVM (for maintaining type-safety) is the `getelementptr` instruction.

3.3.1 Type-safe Pointer Arithmetic with the `getelementptr` Instruction

The `getelementptr` instruction is used to calculate the address of a sub-element of an aggregate data structure in a type-safe manner. Given a pointer to a structure and a field number, the `getelementptr` instruction yields a pointer to the field. Given a pointer to an array and an element number, the instruction returns a pointer to the specified element. In addition to single-level indexing, multiple indexes can be specified at the same time in one `getelementptr` instruction.

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

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

Figure 3.6: C code for complex memory addressing

The example in Figure 3.6 is an example of complex memory access in C, designed to be a concise illustration of the LLVM lexical structure, type system, and the `getelementptr` instruction. The test-case defines two structure types and a function which performs complex indexing.

The LLVM code in Figure 3.7 is a version of the code generated by the C front-end, with LLVM comments added. In addition to illustrating the `getelementptr` instruction, this code shows that LLVM identifiers (type and value names) start with a `%` character (to prevent namespace collisions with reserved words), shows some examples of complicated nested types, and shows an LLVM

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

; Define function 'foo', returning an 'int*', taking an 'ST*':
int* %foo(%ST* %s) {
  ; Perform the indexing...
  %tmp = getelementptr %ST* %s, long 1, ubyte 2, ubyte 1, uint 5, uint 13
  ret int* %tmp         ; Return the computed value
}
```

Figure 3.7: LLVM code for complex memory addressing

function definition. Additionally, it illustrates how useful named types are for hand inspection of code; without symbolic names provided by the C compiler, the types would all be expanded out inline, making them less manageable.

3.3.2 Distinguishing Safe and Unsafe Code: the `cast` Instruction

There are two broad reasons why type conversions may be required in programs: (a) explicit conversions of a value from one type to another, which may or may require manipulating the data (e.g., integer to floating point or signed integer to unsigned), and (b) reinterpreting data of one type as data of another type (e.g., treating data in memory as a linear sequence of bytes instead of an array of integers).

In LLVM, type conversions can only happen in one carefully controlled way: the `cast` instruction. The `cast` instruction converts a value from one type to another. Some examples are illustrated in Figure 3.8.

```
%Y = cast int %X to double     ; requires data conversion
%J = cast int %I to long       ; may require data conversion
%J = cast int %I to uint       ; no data conversion needed
%q = cast int   %pd to double* ; no data conversion needed; may be unsafe
%r = cast void* %pi to QItem*  ; no data conversion needed; may be unsafe
```

Figure 3.8: Examples of casts in LLVM

The cast instruction takes the place of typical sign extension instructions (signed and unsigned types are distinct) as well as integer/floating point conversion instructions. The cast is also used for operations that do not alter data as well, such as converting a signed integer to an unsigned integer of the same size.

Because LLVM is intended as a general-purpose low-level instruction set, it must represent both "type-safe" and "type-unsafe' programs for arbitrary high-level languages. Nevertheless, distinguishing between safe and unsafe operations is important because many memory-oriented optimizations may only be legal for safe programs.

We consider an LLVM program to be *type-safe* if no `cast` instruction converts a non-pointer type to a pointer type or a pointer of one type to a pointer of another type (in other words, no casts *to* a pointer type are allowed). In the example above, the last two cast instructions are unsafe. Such pointer casts are *the only way* that operations of the second type above (that reintepret data in memory) can be encoded in LLVM.

If a program is type-safe by the above definition, type information can be exploited during important analyses such as alias analysis, and data structure reorganization transformations can be safely applied to it[3]. If a program is completely type-safe, its LLVM code can use the `getelementptr` instruction for all pointer arithmetic, without requiring any unsafe casts. For example, given a language with pointer arithmetic, naive compilation can cause type violations. For example, consider the C code in Figure 3.9:

```
int *A = ..., *P = A;
while (P != A+Size) {
  *P = *P + 1;
  ++P;             /* Pointer arithmetic! */
}
```

Figure 3.9: Pointer arithmetic example in C

The pointer arithmetic in the last statement could be compiled initially into the snippet of LLVM code in Figure 3.10.

The `%P2` cast is not type-safe, because an arbitrary value is being cast to a pointer, and although

[3]Note that programs which have "undefined behavior", e.g., by accessing memory that has been `free`'d or using out-of-range array subscripts, are still be considered "type-safe" by our definition. This is appropriate because such behavior does not preclude transformations: the compiler can legally change the behavior of such programs, without having to detect any correctness violations that result from undefined behavior. In a language with stricter safety requirements, such as Java or Fortran 90, the additional "type-safety" checks required by the language should be implemented with explicit LLVM code (for example, conditional branches on the bounds of the array) and then optimized, just as they would be in the low-level representations within standard static compilers.

```
%tmp  = cast int* %P1 to long    ; Convert pointer to integral type for add
%tmp2 = add  long %tmp, 4        ; Add offset to integral value
%P2   = cast long %tmp2 to int*  ; Convert result back to pointer (unsafe!)
```

Figure 3.10: Pointer arithmetic example in LLVM

in this case the result happens to be a valid integer pointer, the compiler cannot know that without further analysis. In the case above, however, the `getelementptr` instruction can be used to directly navigate the array in a type-safe[4] manner (Figure 3.11).

```
%P2 = getelementptr int* %P1, long 1   ; Get pointer to next integer
```

Figure 3.11: Pointer arithmetic example with the `getelementptr` instruction

In practice, many C programs are completely or mostly type-safe according to the above definition, and most unsafe cast operations can be eliminated from such programs through simple transformations. Nevertheless, there are some programs which intrinsically must use unsafe operations (such as casting a specific integer, representing the address of a memory-mapped hardware device, to a pointer), which cannot be converted to use the `getelementptr` instruction. In these cases, the `cast` instruction in LLVM gives critical information about *when* the type system is being violated, improving analyses and allowing for straightforward determination of whether a transformation is *safe*.

3.4 Explicit Memory Allocation and Unified Memory Model

Some of the hardest programs to adequately optimize are memory bound programs that make extensive use of complex data structures on the heap. To better expose memory allocation patterns to the compiler, we have added typed memory allocation instructions to the instruction set. 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 the memory allocated through the `malloc` instruction[5]. The `alloca` instruction is similar to `malloc` except that it allocates memory

[4]This example also shows an instance where out of range array access would not and could not be trapped.

[5]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.

in the stack frame of the current function instead of the heap, and the memory is automatically deallocated on return from the function.

These instructions are essential for preserving the type-safety of our representation[6], and the enable new transformations such as Data Structure Analysis (Section 4.2) and Automatic Pool Allocation (Section 4.3), that would be very difficult without them. An important property of these aggressive techniques is that they are safe even for non-type-safe programming languages, such as C.

The LLVM virtual instruction set is also unique in the manner it handles memory. In LLVM, all addressable objects (stack allocated locals, global variables, functions, and dynamically allocated memory) are all explicitly allocated, giving a unified memory model. Stack allocated locals ("automatic" variables and source-level `alloca()` calls) are all explicitly allocated using the `alloca` instruction. Heap allocated memory is allocated with the `malloc` instruction. Functions and global variables (collectively referred to as "global values") declare regions of statically allocated memory that are accessed though the address of the object (the name of the global value refers to the address).

An interesting effect of always accessing memory objects by their address is that LLVM does not need an "address-of" operator at all. This representation also simplifies memory access analysis since there cannot be implicit accesses to memory. All memory traffic occurs when `load` and `store` instructions execute.

3.5 Function Calls and Exception Handling

LLVM provides two function call instructions, which abstract away the calling conventions of the underlying machine, simplify program analysis, and provide support for exception handling. The simple `call` instruction takes a pointer to a function to call, as well as the arguments to pass in (which are passed by value). Although all `call` instructions take a function pointer to invoke (and are thus seemingly indirect calls), direct calls are easily identifiable. The second function call instruction provided by LLVM is the `invoke` instruction, which is used for languages with destructors to implement exception handling.

[6]The normal C `malloc()` function returns an untyped pointer that must be cast to the appropriate type

LLVM implements a stack unwinding mechanism for “zero cost” exception handling [9]. A “zero cost” exception-handling model indicates that the presence of exception handling causes no extra instructions to be executed by the program when exceptions are not thrown. If an exception is thrown, the stack is unwound, stepping through the return addresses of function calls on the stack. The LLVM runtime keeps a static map of return addresses to exception handler blocks that it uses to invoke handlers during unwinding.

In order to build this static map of handler information, LLVM provides an `invoke` instruction that takes an exception handler label in addition to the function pointer and argument operands of a normal `call` instruction. When code generation occurs, the return address of an invoke instruction is associated with the exception handler label specified, allowing the exception handling/cleanup routine to be invoked when the stack frame is unwound.

The `invoke` instruction is capable of representing high-level exceptions directly in LLVM using only low-level concepts (return address to handler map). This also makes LLVM independent of the source language's exception handling semantics. In this representation, exception edges are directly specified and visible to the LLVM framework, ensuring that all LLVM transformations are correct in the face of exceptions. The example in Figure 3.12 illustrates a case where the `invoke` instruction would be generated by a C++ front-end.

```
{
  Class Object;     // Has a destructor
  func();           // Could throw
  ...
}
```

Figure 3.12: C++ exception handling example

The key thing to note with this very simple example is that C++ guarantees that the destructors of stack allocated objects will be invoked when the block is exited. If an exception is thrown as a result of the `func()` call, the block will be exited, so a handler must be installed to call `Object`'s destructor. Figure 3.13 shows the LLVM code for the example.

The `invoke` instruction associates an exception handler to call if an exception is propagated through the invoked function. In the example, this is used to invoke the destructor of a local object. In the context of the Java language (which does not need to call destructors when unwinding),

```
    ...
    %Object = alloca %Class      ; Stack allocate object
    ; ... call constructor on %Object ...
    invoke void %func() to label %OkLabel except label %ExceptionLabel
OkLabel:
    ; ... execution continues...
    ; ...
ExceptionLabel:
    ; ... call destructor on %Object ...
    call void %rethrow()               ; Rethrow current exception
```

Figure 3.13: LLVM code for exception handling example

the `invoke` instruction is used to unlock locks that are acquired through synchronized blocks or methods. In any language, a `catch` clause would be implemented in terms of an exception destination.

Although we currently do not have a front-end that uses the exception handling support built into LLVM, all of our optimizations and transformations are aware of the exceptional control flow edges and our unit tests work as designed. We also plan to implement the C `setjmp` and `longjmp` calls using this facility.

3.6 Plain-text, Bytecode, and In-memory Representations

The LLVM virtual instruction set is the glue that unifies the system design described in Chapter 2. As such, the effectiveness and ease of use of the system depends on many aspects of the instruction set design. An important feature of the LLVM virtual instruction set is that it is a first class language, complete with a textual format (examples of which have been included in this document), a compressed binary format, and an in-memory format suitable for transformation.

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 Optimizations with LLVM

Low-level representations are notorious for being poor hosts for high-level analyses and transformations. For example, the SGI Pro64 compiler[37] intermediate representation (WHIRL) contains no less than 5 different levels of the representation in order to perform optimizations on the highest level possible. In this high-quality compiler, all interprocedural optimization is done on a very high-level representation, which is effectively a language neutral Abstract Syntax Tree.

LLVM aims to enable high-level transformations at link and post-link time. For many reasons however, it is advantageous to represent code in a low-level form. This chapter describes several aggressive transformations and analyses that are performed on the LLVM representation, showing that an efficient low-level representation, enriched with type information, can support this work. We maintain that a *low-level representation*, with *high-level type information*, **can** support interesting high-level transformations.

Note that all of these analyses and optimizations are actually implemented in LLVM and are robust enough for daily use.

4.1 Simple Analyses and Transformations

Because LLVM is a low-level representation, it is well suited for many traditional transformations and analyses that work on three-address code. Indeed many traditional optimizations have been implemented for this thesis and are in use by the LLVM compilers. Some of the more interesting ones include the following:

1. **Traditional SSA based optimizations:**

   Simple Dead Code Elimination, Aggressive Dead Code Elimination, Global Common Subexpression Elimination, Induction Variable Simplification, Loop Invariant Code Motion, Expression Reassociation, Simple Constant Propagation, Sparse Conditional Constant Propagation, Value Numbering, etc...

2. **Control Flow Graph based optimizations and analyses:**

   Critical Edge Elimination, various forms of Dominator Information, Interval Construction, Natural Loop Construction, Loop Pre-header Insertion, CFG Simplification, etc ...

3. **Interprocedural analyses and transformations:**

   Call Graph Construction, various Alias Analyses, Global Constant Merging, Type Safety Analysis, Dead Global Elimination (both global variables and functions), Inlining, etc ...

In addition to these transformations, LLVM contains two intraprocedural passes, descriptions of which will help illustrate how the LLVM system works together. These relatively unconventional passes are the Instruction Combining and Level Raising passes.

4.1.1 The Instruction Combining Pass

The Instruction Combining Pass is a simple work-list driven pass that operates on the SSA graph of a function, performing peephole optimizations. Because the pass operates on the SSA graph, this simple peephole optimizer is able to perform powerful *global* optimizations. The pass contains many identities such as "`add X,0 = X`", "`sub X,X = 0`", "`and X,-1 = X`", "`mod X,1 = 0`", "`xor X,X = 0`", "`add (add X,1),1 = add X,2`", "`shl X, 64 = 0`", "`phi X, X = X`", etc...

The instruction combining pass also ensures that instructions are in their canonical form, when appropriate. For example, if an `add` instruction has a constant argument, the canonical form has the constant as the second argument. Because the instruction combination is extremely fast, and because is can dramatically simplify the code, many passes can safely assume that their input does not have trivial cases like these to handle, which simplifies their implementation. The instruction combining pass also takes the place of a simple value numbering pass which is able to directly eliminate many of the equivalent expressions that the value numbering pass would identify.

4.1.2 The Level Raise Pass

Most existing compilers retain very little type information for the program once they convert from a high-level code representation (e.g., an AST) to a lower-level representation used for optimizations. Because re-targeting existing compilers is an important part of the overall LLVM strategy, we developed the "level raising" pass to centralize type information reconstruction.

With the level raising pass, compiler front-ends can be modified to simply generate *legal* LLVM code that is simple but low-level and "poorly-typed" (e.g., using explicit byte addressing for structure field accesses). The front-end can use the level raising pass to recover type information (e.g., direct references to structure fields, using the `getelementptr` instruction described in Section 3.3.1). In this way, the program analysis required to recover type information can be shared by many pre-existing compiler front-ends without any integration problems: it is simply another LLVM to LLVM transformation.

The "level raising" pass eliminates `cast` instructions from LLVM code using several different strategies. It assumes that the type information for function interfaces is correct, eliminating as many `cast` instructions possible. We believe that the prototypes for functions are easily accessible from the debug information maintained by the compiler and likely to be correct, even if the compiler is performing aggressive optimizations. Because optimizations may arbitrarily modify the body of the functions, however, debug information is more difficult to use within the body of the function.

Figure 4.1 contains C code which will be used to demonstrate the transformations made by the level raising pass:

```
struct S1 {                                  /* pair of integers */
  int i, j;
};

unsigned foo(struct S1 *s) {
  return s->i*4 + s->j;                      /* access the two fields */
}
```

Figure 4.1: Level-Raise example C code

From this C snippet, the LLVM C front-end outputs the LLVM code in Figure 4.2, which is correct, but poorly-typed. In this example, the strength reduction of the multiply was done by the C front-end, not by LLVM.

In this case, the level raise pass first notices that `%cast215` casts a structure pointer to a pointer to the type of the first element of the structure. This `cast` can be eliminated by simply converting the `cast` instruction into a `getelementptr` instruction of the first field. Similarly, `%s+4` is found equal to `&s->j`, so `cast217` can be raised to use a safe `getelementptr` instruction. Figure 4.3

```
%S1 = type { int, int }                                   ; typedef information

; Function prototype information preserved from "debug" information
uint "foo"(%S1* %s) {
  %cast215 = cast %S1* %s to int*                        ; cast215 = &s->i
  %reg109 = load int* %cast215                            ; Load the s->i field
  %reg111 = shl int %reg109, ubyte 2                      ; Multiply by 4
  %cast214 = cast %S1* %s to ulong                        ; Low-level structure
  %reg213 = add ulong 4, %cast216                         ;   address arithmetic
  %cast217 = cast ulong %reg213 to int*                   ; cast217 = &s->j
  %reg112 = load int* %cast217                            ; Load the s->j field
  %reg108 = add int %reg111, %reg112                      ; Add the fields
  ret int %reg108                                         ; Return result
}
```

Figure 4.2: Level-Raise example raw LLVM code

contains the code after the level raising pass has improved it.

```
%S1 = type { int, int }                                   ; typedef information

int "foo"(%S1* %s) {
  %cast215 = getelementptr %S1* %s, uint 0, ubyte 0  ; cast215 = &s->i
  %reg109 = load int* %cast215                            ; Load the s->i field
  %reg111 = shl int %reg109, ubyte 2                      ; Multiply by 4
  %cast217 = getelementptr %S1* %s, uint 0, ubyte 1  ; cast217 = &s->j
  %reg1121 = load int* %cast217                           ; Load the s->j field
  %reg108 = add int %reg111, %reg1121                     ; Add the fields
  ret int %reg108                                         ; Return result
}
```

Figure 4.3: Level-Raise example LLVM code after level raising

This example has been proven to be type-safe, despite the fact that the input program have very little in the way of type information.

4.2 Data Structure Analysis

Data Structure Analysis was the first completely novel interprocedural link-time analysis developed in the LLVM compiler infrastructure [28]. It makes extensive use of the link-time analysis and type information, and is made more precise through the use of SSA representation.

Data Structure Analysis is a context-sensitive, flow-insensitive heap analysis algorithm which is specifically designed to support *macroscopic data-structure analyses and transformations*. This is a new class of algorithms being developed, which use *instances* of abstract data structure as the scope of the analysis or transformation.

A key strength of Data Structure Analysis is its ability to be used for a variety of different analysis problems. In addition to macroscopic data-structure transformations (such as Pool Allocation, described in Section 4.3), Data Structure Analysis may be used as the base for a variety of traditional analyses as well (such as alias analysis, interprocedural Mod/Ref, "de-virtualization" of function pointers, etc.).

Data Structure Analysis has several key properties that are required to enable transformations such as those mentioned above: *context-sensitivity with cloning*, *field-sensitivity*, and the use of an *explicit heap model*. All these properties, especially the first, are widely considered to incur a high analysis cost [25]. In order to overcome this cost, Data Structure Analysis introduces key simplifications which make it both scalable and extremely fast in practice.

This section describes the Data Structure Analysis algorithm at a very high level, showing how the design of LLVM allows for aggressive interprocedural analyses at link-time. Section 4.2.1 describes the graph representation built by the analysis and provided to clients. Section 4.2.2 describes the algorithm used to construct the data structure graphs. More detailed information can be found in the original paper describing this algorithm [28].

4.2.1 The Data Structure Graph

Data Structure Analysis concisely summarizes a program's memory connectivity patterns and composition by building a Data Structure Graph (DS Graph) for each function in the program. The graph for a function is composed of three pieces of information: a set of nodes with outgoing edges, a mapping of scalars in the function to nodes in the graph, and a list of call sites in the context of the graph. These are described below, in turn.

Figure 4.4 shows a DS graph for the `do_all` function of Figure 4.5 (computed only from local information). We use this as a running example below.

Data Structure Nodes

The DS Graph is a partitioning of the (potentially unbounded) memory objects created during the dynamic execution of the program. Each node in the graph represents a set of memory objects, and edges represent *may-point-to* relationships. Data Structure Analysis uses a unification-based approach to heap modeling (also used by Steensgaard's points-to analysis algorithm [42], for example). For this reason, each field in each node of a graph can only point to a single destination node. If the analysis discovers two different nodes that may be pointed to by the same field, it merges the two nodes together.

Each node is capable of representing objects with multiple distinct outgoing edges, i.e., the points-to information is *field-sensitive*. For example, in Figure 4.4, the node `list` has two fields, with outgoing edges from the first field and incoming edges both to the node and to the second field.

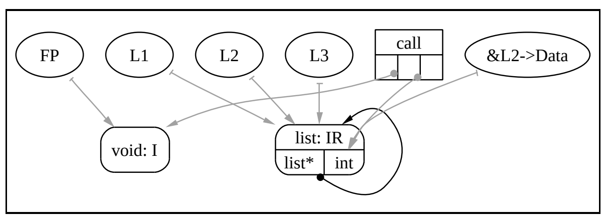


Figure 4.4: Local DSGraph for the `do_all` function

Field sensitivity is an example of a property that is greatly enhanced by the typed LLVM representation. Data Structure Analysis is able to optimistically assume that memory objects are strongly typed, until a violation is noticed. This assumption significantly speeds up analysis because the vast majority of data structures are type safe in practice, even in non-type-safe languages such as C. If a non-type-safe construct is found, the node is "collapsed" down to a single field, thus losing field sensitivity for the node but retaining conservative correctness. In practice, node collapsing is infrequent, even for C programs.

Each node also has a small set of flags that track information about the node. Four of these bits track the classes of memory represented by the node. One bit tracks whether or not the node is "complete", and two bits track mod/ref information for the node.

Data Structure Analysis partitions memory into four different classes of objects: **H**eap-allocated,

```
typedef struct list { struct list *Next; int Data; } list;
int G = 10;
void do_all(list *L1, void (*FP)(int*)) {
  do { L2 = phi(L1, L3);              /* SSA phi node */
       FP(&L2->Data);
       L3 = L2->Next;
  } while(L3);
}
void addG(int *X) { (*X) += G; }
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);
  addGToList(Y);
}
```

Figure 4.5: C code, in SSA form, for Data Structure Analysis example

Stack-allocated, **G**lobal (corresponding to the 3 types of LLVM memory objects), and **U**nknown objects. Unknown memory objects occur when a constant value is cast to a pointer value (for example, to access a memory-mapped hardware device), or when unanalyzable address arithmetic occurs. These cases occur infrequently in portable programs.

Data Structure Analysis tracks the DS Node pointed to by each scalar in the program. A `ScalarMap` maps each pointer-compatible register to the (single) node it may point to, or to NULL. We denote scalars by ellipses in our diagrams (e.g., `FP, L1, L2, L3, &L2->Data` in Figure 4.4). Because LLVM uses SSA form as its representation, this mapping is very simple to maintain.

DS Graphs must correctly represent incomplete programs where some functions are unavailable for analysis. To do this efficiently, each node in the data structure graph contains a bit to indicate if it is "**I**ncomplete". If this bit is set for a node, there may be outgoing edges that are not represented in the graph, or other information such as type or mod/ref information may be missing. If the bit is clear, the node is fully represented.

For example, in Figure 4.4 both memory nodes (labelled `void` and `list`) have the **I** flag set because the pointers from formal arguments `L1` and `FP` imply that those nodes may be modified outside the context of the current function. These **I** flags will be eliminated later using interprocedural analysis.

Because Data Structure Analysis tracks which nodes in the graph may contain incomplete information, the DS Graph is sound regardless of how much information has been incorporated into it. This also dramatically simplifies the construction algorithms presented in Section 4.2.2.

The last two bits tracked by a DS graph node, **M**od and **R**ef, indicate whether or not any of the objects represented by node have been modified or read in the context of the graph. The partitioning of memory objects in the DS Graph provides a natural way to represent this information. In Figure 4.4, the ref bit is set on the list node because the `Next` field is read.

Data Structure Edges

In a DS Graph, edges represent may-point-to information. Because DS Graphs are field sensitive, edges must contain the node they point to as well as an offset into this node. In Figure 4.4, the edges from `FP`, `L1`, `L2`, and `L3` all have an offset of zero, but the edge from `&L2->Data` has a node offset of 8 bytes (which is the size of a pointer in our system).

Call Site Information and Return Values

The DS graph for a function may contain "call nodes" in addition to traditional memory nodes. The presence of a call node indicates that an unresolved function call exists within the current function, which may occur either due to an incomplete program or unfinished analysis. These call nodes directly correspond to `call` instructions in LLVM.

The leftmost field of the call node represents the return value. If a function call returns a pointer value, this field points to the returned memory object or (before interprocedural analysis) an Incomplete node that can later be merged with that object. The second field points to the memory object representing the callee function or functions (an indirect function call may potentially call more than one function). Subsequent fields represent the actual arguments passed at the call site. Figure 4.4 includes an indirect call invoking the function pointed to by the scalar FP, with no return object (since `*FP` returns `void`) and one actual argument.

Finally, if a function returns a pointer type, its graph represents the returned object as a special scalar labeled "returning" with an edge to the object being returned.

4.2.2 Construction Algorithm

Data structure graphs are created in a three step process. First, an intraprocedural phase processes each function in the program and abstracts the behavior of each into a "Local" data structure graph, ignoring the effect of callers and callees. Next, a "Bottom-Up" analysis clones and merges callee graphs into their callers. The final "Top-Down" phase merges caller graphs back into their callees.

We use the example program in Figure 4.5 as a motivating example. It illustrates some of the high-level challenges that Data Structure Analysis algorithm can handle.

Local Analysis Phase

The local analysis phase captures the memory usage behaviors of individual functions without including calling or caller context. The local analysis phase is the only phase of Data Structure Analysis that directly uses the LLVM virtual instruction set, so we will describe it in more detail than the other phases. The top level approach of the algorithm is illustrated in Figure 4.6.

LocalAnalysis(Function F)
  Seed scalar map
  $\forall$ Instruction $I \in F$
    ProcessInstruction(I)
  Mark nodes incomplete Section 4.2.2

Figure 4.6: Local Analysis Algorithm

The analysis starts with an initial conditioning step to seed the "ScalarMap" to include entries for any non-instruction value used by the function of pointer-compatible type. For example, this step is the source of the two **G**lobal nodes in Figure 4.7. The next phase of the analysis is a flow-insensitive linear pass over the program representation. The LLVM version of the `ProcessInstruction` function is shown in Figure 4.8. We describe a few cases in more detail here.

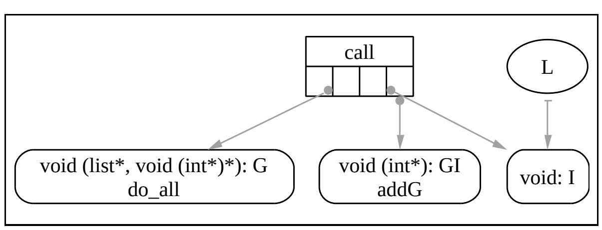


Figure 4.7: Local DSGraph for the `addGToList` function

Allocation sites create a new node with the appropriate memory class bit set. Because the LLVM virtual instruction set allocates all heap and stack memory through the `malloc` and `alloca` instructions, these are very easy to identify (other representations do not distinguish between automatic variables that have their address taken from variables that do not).

Load instructions updates mod/ref information and merge the load source and destination pointers. Return instructions are handled by updating the return value for the current DS Graph (§4.2.1). Each call instruction is represented as a new call site object in the graph, which uniformly represents direct and indirect calls (for example, see Figure 4.7).

ProcessInstruction(Instruction I)
  X `= malloc ...` or X `= alloca ...`:
    mergeEdges(ScalarMap[X], new Node)
    Set **H** *or* **S** *bit in node ScalarMap[X]*
  X `= &`$Y \rightarrow Z$: *(address of struct field)*
    mergeEdges(ScalarMap[X], addOffset(ScalarMap[Y], Z))
  X `= &`Y`[`idx`]`: *(address of array element)*
    mergeEdges(ScalarMap[X], ScalarMap[Y])
  X `= load` Y: *(in C, X = *Y)*
    Set **R** *bit in node ScalarMap[Y]*
    mergeEdges(ScalarMap[X], getLinkAt(ScalarMap[Y]))
  `store` X `into` Y: *(in C, $*Y = X$)*
    Set **M** *bit in node ScalarMap[Y]*
    mergeEdges(ScalarMap[X], getLinkAt(ScalarMap[Y]))
  X `= cast` Y `to` τ: *(in C, $X = (\tau)Y$)*
    mergeEdges(ScalarMap[X], ScalarMap[Y])
    if (not type-safe) collapseNode(ScalarMap[Y])
  X `=` $\phi(Y_1, Y_2, ...)$:
    $\forall Y_i \in Args$: mergeEdges(ScalarMap[X], ScalarMap[Y_i])
  `return` X:
    mergeEdges(ReturnEdge, ScalarMap[X])
  X `= call` Y`(`Z_1`,` Z_2`, ...)`
    CallSite CS = new CallSite
    mergeEdges(ScalarMap[X], retval(CS))
    mergeEdges(ScalarMap[Y], callee(CS))
    $\forall Z_i \in Args$: mergeEdges(ScalarMap[Z_i], CS.getArg(i))
  Otherwise:
    Collapse nodes and set **U** bit for any pointer args

Figure 4.8: ProcessInstruction for the LLVM virtual instruction set

`cast` instructions are used to communicate important information about when the type system is being violated (see Section 3.3.2). When a pointer converting, non-type-safe cast is encountered, the operand node is folded. This folding operation discards field sensitivity in order to retain conservative correctness. The explicit exposure of this operation in the LLVM virtual instruction set makes this loss of type information explicit.

For other instructions involving a pointer-compatible operand or result, the local analysis phase sets the "**U**nknown" bit and collapse the node to indicate that something untraceable occurred.

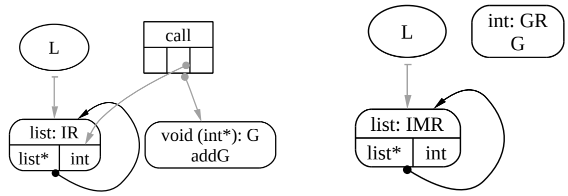
The final step in the Local graph construction is to calculate which data structure nodes are complete and which are incomplete. For a Local graph, any node reachable from a formal argument, global, passed as an argument to a call site, or returned by a call site is marked as incomplete.

Bottom-Up Analysis Phase

The Bottom-Up (BU) analysis phase creates a graph for each function in the program, concisely summarizing the total effect of calling that function (imposed aliases and mod/ref information) without any calling context information. It computes this graph by cloning the Bottom-up graphs of all *known* callees into the caller's Local graph, merging nodes pointed to by corresponding formal and actual arguments.

In the DS graph representation, the DSCallSite list maintained by each DS graph implicitly defines the known edges of the call graph. The Bottom-Up analysis phase uses Tarjan's linear-time algorithm to identify Strongly Connected Components (SCCs) in the call graph defined by the DSCallSite list. Tarjan's algorithm identifies SCCs in post-order, directly providing the Bottom-Up order that this phase requires. While traversing the call graph in post-order, the Bottom-Up Analysis phase clones each called graph into its caller, resolving arguments and eliminating call sites.

Handling function pointers and external functions requires that we restrict the post-order traversal to only walk call-sites which target "complete" nodes (§4.2.1). As graphs are cloned into their caller, the unresolved call nodes will be copied as well. Such an unresolved call may become resolved (because the function passed to a function pointer argument becomes known). This allows the indirect call to be resolved by inlining the callee's BU graph into the graph of the function where the call site became resolved. For example, in Figure 4.9, the indirect call site in `do_all` is



(a) After cloning `do_all` (b) Finished

Figure 4.9: Bottom-Up DS Graphs for `addGToList`

resolved while processing `addGToList`. Note that the BU Graph containing the original call site (`do_all` in the example) will not have the call resolved during the BU pass.

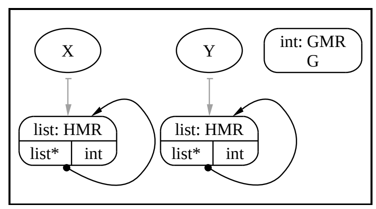


Figure 4.10: BU DSGraph for the `main` function

The graph of Figure 4.10 shows the Bottom-Up graph calculated for the `main` function of our example. This graph demonstrates how the combination of context sensitivity with cloning can identify disjoint data structures, even when complex pointer manipulation is involved.

Top-Down Analysis Phase

The Top-Down analysis pass is used to propagate information from callers to callees. The goal of this phase is to construct a graph for each function which describes all of the possible contexts the function is invoked in.

The Top-Down construction phase is almost the exact inverse of the Bottom-Up construction phase. We traverse the inverse call graph (the call graph computed by the Bottom-Up traversal with all edges inverted), using the Tarjan SCC identification algorithm to handle SCCs in the same way as the Bottom-Up phase. Instead of inlining callee graphs into the caller graph in the **ProcessSCC** function, the Top-Down pass inlines the caller graph into each of its callees.

The primary distinction between the Bottom-Up and Top-Down construction phases is that the incomplete node marker does not mark argument nodes as being incomplete if all callers can be identified by the analysis, which is safe once all calling contexts are taken into consideration. With incoming arguments "complete," it is safe to decide that two incoming pointers are never aliased, for example.

4.2.3 Complexity, Results, and Applications

Despite the capabilities of Data Structure Analysis for identifying complex recursive data structures, the analysis time complexity is only $\Theta(s^2)$ in the worst case (where s is the size of the largest SCC in the program), and $\Theta(n)$ in the common case (where n is the number of functions in the program) [28].

In addition to a low asymptotic complexity, Data Structure Analysis is efficient in practice. Lattner and Adve [28] show that Data Structure Analysis is quite efficient and scalable for all programs tested (the ptr-dist, Olden, and some SPECINT2000 benchmarks, including programs of up to 71,364 lines of C code), requiring only up to 1.34 seconds to analyze the largest program while building local, bottom-up and top-down graphs. The memory requirements of Data Structure Analysis are also modest, using only 4.8MB of memory to represent the results for the largest program.

One of the key features of the Data Structure Analysis algorithm is that it may be used for a variety of different applications. The top-down graph is directly useful as a memory object disambiguator (implementing alias analysis). Additionally, the mod/ref information captures by the nodes of the graph may be used as the basis for a simple Interprocedural Mod/Ref implementation. This implementation needs no additional interprocedural analysis to compute a context-sensitive, flow-insensitive result, making it both very simple and very powerful.

There are many other applications of data structure analysis, ranging from accurate call graph construction to an aggressive interprocedural transformation known as Automatic Pool Allocation.

4.3 Automatic Pool Allocation

Many researchers have demonstrated the value of pool allocating data structures [14, 20, 43], but *fully automatic* pool allocation is a challenging problem. Here we describe a simple algorithm, developed by Lattner and Adve[30], which use the LLVM infrastructure, for fully automatic pool allocation of C programs. This algorithm uses the Data Structure Graph to ensure safety of transformation. Automatic Pool Allocation is an example of a *macroscopic* transformation, which works on entire data structures at a time. It fundamentally requires the strong interprocedural capabilities of LLVM to be effective.

```
void ProcessLists(unsigned N) {
  List *L1 = malloc(sizeof(List));
  List *L2 = malloc(sizeof(List));
  unsigned i;
  for (i = 0; i != N; ++i) {      /* populate lists */
    addList(L1, malloc(sizeof(Patient)));
    addList(L2, malloc(sizeof(Patient)));
  }
  useLists(L1, L2);   /* Use lists */
}
```

Figure 4.11: Source for `ProcessLists` function

The `ProcessLists` function in Figure 4.11 will be used to illustrate how the pool allocation works. The most important part of the example is that it creates two disjoint doubly linked-lists of objects, using a common creation function and a common traversal function. Figure 4.12 shows the Bottom-Up data structure graph for the example.

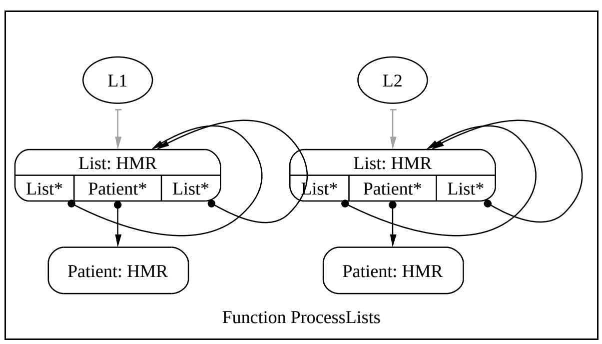


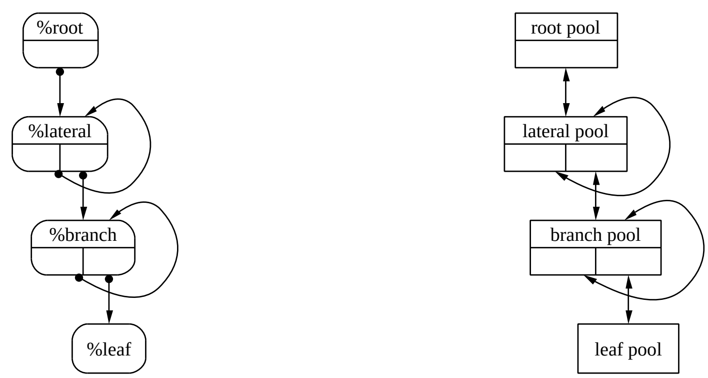
Figure 4.12: Bottom-up DS Graph for `ProcessLists` example

Section 4.3.1 describes the runtime support used by programs which have been transformed to use pool allocation, Section 4.3.2 describes how candidate data structures are identified for pool allocation, and Section 4.3.3 describes how the program is transformed to use pool allocated data structures.

4.3.1 Runtime Support for Pool Allocation

We designed a simple pool allocation runtime library with four external functions (`poolinit`, `pooldestroy`, `poolalloc`, `poolfree`), and one data type (the pool descriptor). We transform the program to pass the pool descriptors into functions that must allocate or free nodes from or to the pool. In this way, the pool descriptor is always available where it is needed.

Our pool allocator assumes that a memory pool consists of uniformly sized objects, but it can allocate multiple consecutive objects if needed (for arrays of objects). When pool allocating a complex data structure (for example, the main data structure for the `power` benchmark from the Olden suite, shown in Figure 4.13(a)) each data structure node in the graph is allocated from a different pool in memory. This simple heuristic groups memory objects together of the same type, which works well for tree nodes, linked lists, and other heavily recursive structures. In the `power` benchmark example, there are four memory pools, each corresponding to a level of a heterogeneous "tree" structure (each level is a linked list of nodes).



(a) Data structure graph (b) Pool descriptor graph

Figure 4.13: Main data structure for `power` benchmark

In addition to bookkeeping information for the pool allocator runtime, the pool descriptors are also augmented to include pointers to the other pool descriptors in the data structure, forming a graph isomorphic to the data structure graph (but accessible at runtime and including back-edges as well as forward edges). For the `power` benchmark, this graph is shown in Figure 4.13(b). By using this graph, the runtime can locate all of the memory blocks allocated to a data structure by traversing the pool descriptors for the data structure.

4.3.2 Identifying Candidate Data Structures

In order to pool allocate a data structure, we must detect the bounds on the lifetime of the data structure (to allocate and delete the pools themselves) and determine whether it is *safe* to pool-allocate the data structure. We use the data structure analysis graph for both purposes.

Using the data structure graph, we detect data structures whose lifetimes are bound by a function lifetime, allowing us to allocate the pool on entry to the function and deallocate it when the function returns. Automatic Pool Allocation identifies these candidates by scanning the functions in the program, inspecting the Bottom-Up graph for each function.

The lifetime of a data structure is contained the current function if the data structures subgraph would be unreachable without the edges due to the scalar pointer map (i.e., no globals point to the structure, and it is not returned from the current function). This escape analysis (which is similar to the points-to escape analysis of [45]) is a conservative, but effective, heuristic for the approximation of data structure lifetime. We refer to the function whose lifetime bounds the lifetime of the data structure as the "root" function, because it is the root of a subtree of the call graph that needs to be modified to handle pool allocation.

It is safe to convert a data structure to use pool allocation if Automatic Pool Allocation can prove that the data structure is used in a type-safe way. Using the Data Structure Analysis graph for the function, a data structure is type-safe if none of its nodes are collapsed, and none of the nodes are incomplete (indicating that the node may be collapsed, but it is not known in this context). This identification algorithm is shown in Figure 4.14.

PoolAllocateProgram(program $Prog$)
  $\forall$ function $Fn \in Prog$
    $\forall$ disjointdatastructure $DS \in$ BUDSGraph(Fn)
      if CollapsedNodes(DS) $\bigcup$ IncompleteNodes(DS) $= \emptyset$ then
        if $\neg$escapes(DS) then
          PoolAllocate(Fn, DS)

Figure 4.14: Candidate identification algorithm

4.3.3 Transforming Function Bodies

Once a data structure is selected for pool allocation, the root function must be modified to allocate pool descriptors representing the various nodes in the subgraph. We insert code to stack-allocate a pool descriptor, initialize the pool descriptor on entry to the function, and destroy the pool descriptor (and all book-keeping information associated with the memory pool) at the exit nodes of the function.

Once the pools are created, the body of the root function (and all functions it calls which have access to the data structure being processed) must be transformed to use `poolalloc` and `poolfree` calls instead of `malloc` and `free` LLVM instructions. To do this, the pool descriptor must be passed into called functions so that they are available for the eventual `poolalloc` and `poolfree` calls. The algorithm is shown in Figure 4.15.

PoolAllocate(function $RootFn$, datastructure DS)
  $Worklist = \{RootFn\}$
  $\forall$ function $Fn \in Worklist$
    $\forall$ instruction $I \in$ Instructions(Fn)
      if UsesDataStructure(I, DS) then
        if IsMallocOrFree(I) then
          ConvertToPoolFunction(I, DS)
        elseif IsCall(I) then
          AddPoolArguments(I, DS)
          $Worklist = Worklist \bigcup$ CalledFunction(I)

Figure 4.15: Function transformation algorithm

The transformation iterates through a work-list of functions to process, transforming each function until the work-list is empty. Initially the work-list is seeded with the root function, and it is expanded whenever a call to an untransformed function is encountered. The body of a function is transformed according to the following rules:

- `malloc` and `free` operations referring to the pool allocated data structure are changed into calls to the `poolalloc` and `poolfree` library functions.

- Function calls which take a pointer into the data structure as an argument, or return a pointer that is part of the data structure, are modified to pass the pool descriptor of the data structure into the called function. If the function has not already been processed, it is added to the transformation work-list.

```
void ProcessLists(unsigned N) {
  PoolDescriptor L1PD, L2PD, P1PD, P2PD;
  List *L1, *L2;   unsigned i;
  poolinit(&L1PD, sizeof(List));
  poolinit(&L2PD, sizeof(List));
  poolinit(&P1PD, sizeof(Patient));
  poolinit(&P2PD, sizeof(Patient));

  L1 = (List*)poolalloc(&L1PD, 1);
  L2 = (List*)poolalloc(&L2PD, 1);

  /* populate lists */
  for (i = 0; i != N; ++i) {
    addList_pa(L1, &L1PD, poolalloc(&P1PD, 1));
    addList_pa(L2, &L2PD, poolalloc(&P2PD, 1));
  }
  useLists_pa(L1, &L1PD, L2, &L2PD);
  pooldestroy(&L1PD); pooldestroy(&L2PD);
  pooldestroy(&P1PD); pooldestroy(&P2PD);
}
```

Figure 4.16: Source for pool-allocated `ProcessLists`

The transformed `ProcessLists` function (Figure 4.16) allocates four memory pools, one for each data structure node in the two disjoint data structures in it. The `addList` function is transformed similarly.

4.4 External Uses of the LLVM Infrastructure

LLVM is designed to support a wide variety of compiler research with many different focuses and needs. One of the best ways to evaluate this capability is to see how well people other than the author use the system. LLVM has been fortunate to have many projects use it in a short amount of time. These projects are briefly described below, hi-lighting the features of LLVM which make them viable.

4.4.1 Ensuring Code Safety without Runtime Checks

Kowshik, Dhurjati and Adve describe a language called Control-C which is designed to provide code safety and correctness guarantees *entirely* through static analysis [27] (as opposed to run-time checks). Control-C a subset of C, which includes heap allocation, pointers, and most other difficult-to-analyze features of the C language, although it elides some features that are typically not used by real-time control applications (which is the target of the work). Once code has been certified to be safe to execute by a compiler using their analysis, it can be installed in the field without fear of it corrupting the state of the control system. This safety is implemented though strong static analyses, eliminating the need for runtime overhead imposed by dynamic safety checks.

This work uses the LLVM system in a variety of ways. It heavily relies on the link-time interprocedural capabilities of the system to perform a flow-sensitive, context-sensitive analysis of array bounds constraints. Various SSA properties of the LLVM virtual instruction set make the analysis more efficient, and the data structure graphs (described in Section 4.2) are used to analyze and evaluate memory references for safety properties.

4.4.2 Program Control Language (PCL)

Ensink, Stanley, and Adve describe a framework for language support to ease development of adaptive applications, named the Program Control Language (PCL) [17]. PCL supports adaptations in distributed applications by separating the adaptive logic from the underlying distributed program, providing an abstraction of program behavior which can be used to reason about and specify adaptation operations, and provides high-level mechanisms for monitoring and adapting program behavior.

Conceptually, PCL allows programmers to change the behavior of an application at runtime by modifying the *static task graph* of the application, for example, by adding and removing tasks and edges. The static task graph provides a *global* view of the entire distributed computation to every participating process. This allows any process to modify the behavior of any other process simply by modifying parts of the static task graph through high-level language extensions and without invoking any explicit communication. The PCL compiler and runtime library hide the complexities of modifying the task graph and performing remote communication.

PCL is built using the LLVM infrastructure, which provides extensive interprocedural analysis capabilities as well as high-level information about program semantics. Future work involves automatic or semi-automatic extraction of the task graph from a program. Data Structure Graphs, along with the control flow graph, dominance, control dependence and call graph information is required for successful analysis of this high-level information. Because whole-program analysis is required for this work, link-time analysis is essential to the work.

4.4.3 Advanced Compilers Class

LLVM served as the host compiler infrastructure for the University of Illinois Advanced Compilers (CS426) class in Fall 2002. All students were required to use LLVM to write a simple global transformation (Scalar Replacement of Aggregates) and to complete a group project.

Sample projects include multiple implementations of SSA Partial Redundancy Elimination [10], several implementations of Anderson's Alias Analysis [3] with Offline Variable Substitution [36] and Online Cycle Elimination [18], and a framework for incremental recomputation of interprocedural data-flow problems [7]. Other groups researched more open ended problems, such as using Data Structure Graphs to introduce static memory management for programs with explicit allocation and deallocation.

5 Evaluating the Infrastructure

Evaluating an infrastructure is difficult: there are very few aspects that can be used to directly evaluate the quality or maturity of the infrastructure. Infrastructures exist solely to enable interesting applications of the infrastructure, not as an interesting application itself. Despite this, there are some important qualities that make infrastructures more successful than others. This chapter attempts to evaluate and quantify several aspects of the LLVM compiler infrastructure in order to provide an idea of the effectiveness of LLVM in the role of compiler infrastructure. The qualities we attempt to estimate are Maturity, Productivity, and Performance.

5.1 Evaluating the Maturity of LLVM

One simple but extremely limited way to evaluate the maturity of a code base is to count the number of lines of code it contains. This metric is useful for establishing the overall size of the project but is prone to potentially significant problems (insignificant details, such as coding conventions, can dramatically influence the numbers). The LLVM compiler infrastructure is written almost exclusively in high-level C++ code, making extensive use of the Standard Template Library and other modern features of the language, so these numbers are a conservative estimate (compared to projects that do not use these features).

At the time of this writing, the LLVM CVS tree contains over 100,000 lines of code (including whitespace, comments, and HTML documentation), not counting the test framework or automatically generated code. Counting Source Lines of Code (code without whitespace, comments, or HTML) yields the information contained in Table 5.1[1]. Note that these numbers only count code in the LLVM CVS repository, which does not include the C front-end.

Another reasonable metric for evaluating an infrastructure is the amount of documentation currently available describing it. This is an important metric, because documentation is necessary for external developers to use the infrastructure effectively. For documentation, the LLVM CVS

[1]Counted by David A. Wheeler's "SLOCCount" tool, available from `http://www.dwheeler.com/sloccount/`

Source Language	SLOC	SLOC %
C++	67194	96.14%
Bison	1578	2.26%
ANSI C	796	1.14%
Flex	323	0.46%

Table 5.1: Source Lines of Code (SLOC) in LLVM Infrastructure

tree contains 8637 lines of HTML documentation (counted with 'wc'), extensive in-source comments describing the various subsystems, and extensive documentation automatically extracted from the source code by the doxygen tool. For additional developer support, LLVM has a web page[31] and several mailing lists.

Although SLOC and documentation can give some indication about the maturity of the infrastructure, by far the best indicator is how LLVM has been used. Chapter 4 describes several ways the LLVM infrastructure is being used today, both by the author and external contributors. These uses show both the capabilities of the infrastructure as well as the features the infrastructure is able to offer to new developers. Of particular note is the fact that LLVM has successfully been used as the host infrastructure for an advanced compilers class (Section 4.4.3). Students tend to be much less forgiving than researchers about poor quality of implementation, lack of documentation, buggy implementation, or poor extensibility. LLVM worked quite well, providing another measure of maturity.

5.2 Evaluating Productivity with LLVM

The LLVM infrastructure provides a solid foundation for research and development as well as teaching. Productivity is very hard to quantify without a detailed study, but some indicators may provide a reasonable argument that LLVM is a productive environment to work in.

The first indicator that we use shows the numbers of SLOC that are required to implement several well known compiler transformations in LLVM. Statistics for four different scalar optimizations are provided in Table 5.2.

These numbers are quite modest considering the capabilities of the individual transformations: The ADCE transformation makes use of the Dominator Tree and Control Dependence Graph provided by LLVM to optimistically delete basic blocks. The GCSE transformation is implemented

Transformation	**SLOC**	**Raw LOC**
Aggressive Dead Code Elimination (ADCE)	203	387
Global Common Subexpression Elimination (GCSE)	129	263
Loop Invariant Code Motion (LICM)	117	223
Sparse Conditional Constant Propagation (SCCP)	310	546

Table 5.2: Source Lines of Code (SLOC) for well-known compiler transformations

in terms of the abstract Value Numbering interface. This Value Numbering interface enables GCSE to automatically make use of alias analysis to disambiguate load/store aliases, allowing it to remove redundant load instructions. The LICM pass makes use of Alias Analysis to disambiguate memory references, allowing it to hoist memory access instructions out of loops. Note also the difference between the SLOC and Raw LOC columns. The difference is due to extensive comments describing the high-level algorithms.

The second productivity indicator is the dramatic rate of progress that LLVM has made in the two years of its development. Despite the fact that the optimizer, infrastructure core, and many other parts of the infrastructure have been almost completely developed by a single programmer, it is a very capable system and is able to support interesting research.

There are several reasons that LLVM is a productive environment: it is written in a high-level language with modern programming techniques; the interfaces to the infrastructure are simple, orthogonal, and stable over time; and the infrastructure makes use of several hundred regression and feature tests to document implemented features and to ensure that bugs stay fixed. Documentation and comments in the code assist developers as they are new to LLVM.

The most important reason for high productivity in the LLVM framework, however, is the LLVM virtual instruction set itself. Because the LLVM IR is a very simple representation, free of complicated special cases and strange behavior, code in transformations and analyses are simplified. An additional advantage of the LLVM virtual instruction set is that it has a well-defined textual representation that may be used to visualize exactly what a pass does, and to construct test-cases for passes manually in LLVM. This makes writing regression tests very simple: simply specify the pass to run, the LLVM input code, and the expected output.

5.3 Evaluating the Performance of LLVM Infrastructure

Despite the fact that LLVM is a strong research infrastructure, care has been taken to ensure that LLVM remains efficient. This is important for clients such as the runtime optimizer, but is also a reasonable "quality of implementation" metric.

As with maturity and productivity, measuring the performance of an infrastructure itself is hard. For this reason, we will show how several well known optimizations are efficient as implemented in the LLVM infrastructure. Since these optimizations *depend* on the infrastructure for a variety of low-level needs, their performance is indicative of the infrastructure efficiency.

		Unoptimized			Optimized		
Benchmark	**#LOC**	**#Inst**	**#BB**	**#Fn**	**#Inst**	**#BB**	**#Fn**
`254.gap`	71k	345,328	25,393	1421	129,153	21,659	885
`255.vortex`	67k	246,844	18,515	965	62,712	13,345	688
`300.twolf`	20k	168,892	7,083	335	46,912	6,071	310

Table 5.3: Static statistics for large SPECINT2000 benchmarks

Table 5.3 describes the three largest SPECINT2000 benchmarks currently compatible[2] with the LLVM C compiler. These statistics describe the number of lines of C code for each benchmark, along with two sets of columns describing the size of the LLVM representation: before and after optimization by the LLVM static optimizer. The dramatic difference in size before and after optimization clearly illustrates the value of performing aggressive optimization at compile-time, instead of performing all optimization at link-time.

Table 5.4 contains data describing the efficiency and effectiveness of four well-known optimizations when run on the `254.gap` benchmark. Because this benchmark is the largest (consisting of 71,364 lines of C code), it was chosen as a representative for more detailed analysis. For these tests, the LLVM infrastructure was compiled at optimization level -O3 by the GNU G++ 3.2 compiler, and tests were timed on a 1.7GHz AMD 2100+ processor.

The raw input to each pass is the entire benchmark linked together, but with no prior optimizations performed (other than what the C front-end does automatically), in order to stress test the optimization. This configuration corresponds to the unoptimized column from Table 5.3. The

[2] The LLVM C front-end is missing minor pieces of functionality preventing it from compiling all benchmarks. Specifically `setjmp`/`longjmp` support, and the ability to define the *body* of a varargs function are missing.

Transformation	# Changes	Time	Time/Inst
ADCE	3843/229	1.42 sec	4.112 μsec
GCSE	90105	24.06 sec	69.673 μsec
LICM	55798	0.90 sec	2.606 μsec
SCCP	57337	6.85 sec	19.836 μsec

Table 5.4: Performance of several optimizations on the `254.gap` benchmark

C front-end does very little optimization by itself (relying on the static optimizer to clean up the code), so the input code is almost completely unoptimized.

From the results in Table 5.4, we see that the infrastructure is quite efficient and effective. Despite the fact that the input is very large, each pass performs quite well due to a clean implementation and a sparse SSA based approach. Additionally, all four optimizations are quite effective at deleting instructions. GCSE, LICM, and SCCP are able to delete tens of thousands of static instructions from the program as they run. ADCE is able to delete 3843 individual instructions and 229 basic blocks worth of instructions as it runs (even though the C front-end does not output anything trivially dead).

Benchmark	ADCE		GCSE		LICM		SCCP	
`254.gap`	1.42s	3843/229	24.06s	90105	0.90s	55798	6.85s	57337
`255.vortex`	1.39s	125/91	18.75s	65532	1.23s	16053	6.91s	52884
`300.twolf`	.26s	161/98	2.45s	52002	.56s	62993	.83s	20663

Table 5.5: Transformation timing and effectiveness results for large SPECINT2000 benchmarks

Table 5.5 contains data for all three large SPEC benchmarks. From this table, we can see that the `254.gap` and `255.vortex` benchmarks are similar with respect to optimization times, but the `300.twolf` benchmark is much faster. We believe `300.twolf` is optimized much more quickly because the program representation and intermediate data structures for transformations fit comfortably into the cache of the processor, whereas the larger benchmarks suffer from capacity misses. These transformations have not been optimized in particular, so their cache usage could probably be improved by a significant margin.

One thing that is important to note is that this test is designed as a *stress test*, not as a typical application of one of these optimizations. In practice, previous optimizations will have greatly reduced the size of the program before the passes execute. Because the static optimizer would have been run on each of the translation units as they are compiled, the resulting program is greatly reduced before link time. In the case of these benchmarks, the static optimizer reduces the input to the linker by 2.67x, 3.94x, and 3.60x respectively.

5.4 Evaluation Summary

Clearly, LLVM has progressed a long way in the first two years of its development. As LLVM is becoming a commercial grade research compiler, it is an increasingly attractive for new research and development. As development continues, we expect LLVM to continue to grow, gaining new capabilities and possibilities.

6 Related Work

LLVM is clearly related to many different projects in many different ways. This Chapter identifies and evaluates some of the most important work related to LLVM. Because the field of compilers is so broad and there is so much prior art, only a small fraction of related work may be included here.

6.1 Compiler Infrastructures

Many compiler infrastructures are available in the research world, targeting a variety of different problems ranging from language support, mid-level optimization, and low-level code generation issues. Because LLVM is targeted at mid-level interprocedural optimizations, we briefly examine the SUIF compiler infrastructure and the SGI Open64 compiler, which both excel in these areas.

The SUIF compiler infrastructure [46, 47] is perhaps the most influential compiler for interprocedural optimization. SUIF is part of the National Compiler Infrastructure project and has been used for an amazing variety of research projects. It is built around a source-to-source translator which uses a very-high-level AST representation. SUIF has been used for powerful interprocedural and profile driven transformations.

The primary drawback to the SUIF system is that it is very slow, because of a very general but also very large AST representation. An additional problem of using an AST representation is that new front-ends cannot be added without adding new node types to the AST. Extending the AST requires existing modules be updated to work with the extensions, which makes it difficult to add new features, and difficult to support older code. For this reason, many important features (such as exceptions) are still not supported in the official SUIF distributions.

The SGI Open64 compiler [37] is an outgrowth of SGI's high-quality commercial compiler projects. As such, it is an industrial strength compiler, with many robust compiler optimizations built in, including interprocedural and profile driven transformations. The compiler uses an intermediate representation named WHIRL which represents code in five different levels (language specific through machine specific), using a continuous lowering system to transform from language specific trees to machine specific code. Optimizations may be performed at the "appropriate" level for the analysis or transformation.

Like SUIF, the SGI Open64 compiler performs interprocedural optimization on a high-level AST representation, deferring most of compilation to link-time. Additionally, the strong profile-driven optimization framework depends on the traditional five stage model, which leads to a number of problems, as described in Section 1.1.3. The multi-level representation is also detrimental to modularity: phases are designed to work on a particular level (or set of them), so they cannot be freely interchanged.

In contrast to these two systems, the LLVM system uses a low-level representation with language-independent types. Because the representation is a simple low-level three-address code representation, it is compact, regular and does not need to be modified to support new front-ends. Using a low-level representation also allows most optimization to happen at compile time, instead of deferring most compilation to link-time as these systems do. Although these systems both provide much more high-level information to the link-time optimizer than LLVM, we maintain that this information adds little value over the LLVM representation. Additionally these systems provide so much information that a runtime and offline reoptimization system is infeasible.

6.2 High-Level Virtual Machines

In recent years, interest in high-level virtual machine technology has exploded. Virtual machines have been available for decades, perhaps starting with the original Pascal P-Code interpreter. Today, many language-level virtual machines, such as SmallTalk [21], Self [44], Java [22], and the Microsoft's Common Language Runtime [32] are available to host their corresponding language (or set of languages in the case of CLR). These platforms offer a large number of security and platform independence features, and tend to use dynamic compilation to achieve acceptable performance.

Note that these systems all represent a program in a much higher (and language specific) representation than LLVM does. Although this presents a lot of high-level information to the runtime optimizer, it also makes it largely impossible for the static compiler to do a meaningful amount of optimization before runtime. As mentioned in Section 1.1.2, this requires the runtime optimizer to perform many mundane optimizations at runtime just to get acceptable code, which makes aggressive optimizations even more costly.

On the other hand, these systems offer features that LLVM does not currently, including security and guaranteed portability. LLVM bytecode is completely portable if the input program is type-safe, but details about the endian-ness and pointer size configuration of the C compiler can leak into LLVM bytecode for non-type-safe programs. Because these systems do not support non-type-safe programs at all, this is a small issue. Another strength of LLVM is that it does not require a specific object model, set of exception semantics, or any other high-level language feature from a front-end. This makes it very flexible, but a set of inter-language conventions would need to be defined to allow code produced by different front-ends to communicate (an LLVM Application Binary Interface, or ABI).

6.3 Intermediate Representations

The LLVM virtual instruction set is an important part of the LLVM system, which determines how many of the larger components work together. Among other properties, the LLVM virtual instruction set is a strongly typed SSA based representation which is a suitable target for any source language. Note that the LLVM virtual instruction set is both an on-disk format as well as an in-memory format used for transformation. This distinguishes it from a variety of work which aims to make front-end creation simpler (e.g. [26]).

A lot of work has been done in the field of typed intermediate representations. Functional languages often use strongly typed intermediate languages (e.g. [38]) as a natural extension of the source language. The Typed Assembly Language [33] project, focuses on using type information to prove program safety. The SafeTSA [1] representation is a combination of type information with SSA form, also focusing on safety properties.

In contrast to this extensive work, the LLVM virtual instruction set does not use type information to prove safety properties of the input program. In fact, a feature of LLVM is that it allows representation of arbitrary programs generated by a C front-end, even the worst behaved. An interesting feature of LLVM, however, is that it is very simple to detect when type violations occur, as described in Section 3.3.2. In general, the LLVM virtual instruction set is designed for maximal performance while providing important high-level information to post-link optimizers.

A number of attempts have been made to make a unified, generic, intermediate representation. The goal of these projects has been to reduce the amount of effort required to create a new language or microprocessor. These projects have largely failed, ranging from the original UNiversal Computer Oriented Language [41] (UNCOL), which was discussed but never implemented, to the more recent Architecture and language Neutral Distribution Format [12] (ANDF), which was implemented but ultimately failed.

LLVM is much less ambitious than any of these projects. The primary difference between the LLVM virtual instruction set and these projects is, again, the level at which they represent programs. These unified intermediate representations attempt to describe languages at the AST level, implying that they must include features from all possible source languages and preserve information that might possibly be used by a target machine. Instead, LLVM approaches the problem the same way that a microprocessor does: implement generic low-level features that any language can be mapped down to. All high-level concepts can be implemented in least-common denominator forms, and in some ways, LLVM simply appears as a very strict RISC architecture to a front-end.

7 Conclusions

This thesis describes the design for an aggressive *multi-stage* optimizing compiler. This compiler is built with the idea that a *low-level* representation with *high-level* type information is powerful enough to perform aggressive link-time and post-link optimizations. To support this claim, we presented several example transformations which require high-level information in order to be effective, and must be performed at link-time (or later) for full effectiveness. Operating on a low-level representation allows the compiler design to be efficient enough for practical use, and allows many traditional optimizations to be implemented in a simple and efficient manner.

The LLVM virtual instruction set is key to the overall design of the LLVM compiler system. We describe some of the novel aspects of the representation, including strong type information, built-in support for low-level exception handling constructs, SSA form, and explicit memory allocation support. Together, these features and others permit the development of a variety of techniques, including novel *macroscopic data structure* analyses and transformations.

Profile-guided optimization is an important family of techniques for extracting maximum performance from a given application. We show how the LLVM compiler infrastructure can obtain highly accurate profile information *in the field*. This profile information may then be used for a variety of transformations either at run-time or scheduled during idle time on the machine.

One important contribution of this thesis is an implementation of this design. The LLVM compiler infrastructure is a mature system, already supporting the development of novel analyses and transformations. LLVM has a clean system design and extensive documentation, making it a natural match for teaching topics in advanced compilers.

Although the LLVM compiler infrastructure has been a quiet success in the first two years of its development, the exciting part is watching it grow. Every day, LLVM is gaining new features, being applied to new problems, and expanding into new areas. I look forward to seeing what LLVM will look like two years from now: at the rate it is growing, it seems that anything is possible.

References

[1] W. Amme, N. Dalton, M. Franz, and J. ery. Safetsa: A type safe and referentially secure mobile-code representation based on static single assignment form. In *ACM 2001 SIGPLAN Conf. on Prog. Lang. Design and Implementation*, June 2001.

[2] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, D. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone? Technical Note 1997-016 Digital Equipment Corporation Systems Research Center, Palo Alto, CA, July 1997.

[3] L. O. Anderson. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[4] Andrew Ayers, Stuart de Jong, John Peyton, and Richard Schooler. Scalable cross-module optimization. *ACM SIGPLAN Notices*, 33(5):301–312, 1998.

[5] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. pages 1–12, 2000.

[6] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57. IEEE Computer Society Press, 1996.

[7] Michael Burke and Linda Torczon. Interprocedural optimization: eliminating unnecessary recompilation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(3):367–399, 1993.

[8] Brad Calder, Peter Feller, and Alan Eustace. Value profiling. In *International Symposium on Microarchitecture*, pages 259–269, 1997.

[9] David Chase. Implementation of exception handling, Part I. *The Journal of C Language Translation*, 5(4):229–240, June 1994.

[10] Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on ssa form. In *Proceedings of the 1997 ACM SIGPLAN conference on Programming language design and implementation*, pages 273–286. ACM Press, 1997.

[11] R. Cohn, D. Goodwin, and P. Lowney. Optimizing Alpha Executables on Windows NT with Spike. *Digital Technical Journal*, 9(4), 1997.

[12] ANDF Consortium. The Architectural Neutral Distribution Format, `http://www.andf.org/`.

[13] IBM Corporation. Xl fortran: Eight ways to boost performance. White Paper, 2000.

[14] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 262–275, New York, NY, 1999.

[15] 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*, pages 13(4):451–490, October 1991.

[16] Kemal Ebcioglu and Erik R. Altman. DAISY: Dynamic compilation for 100In *Int'l Symp. on Computer Architecture*, pages 26–37, 1997.

[17] Brian Ensink, Joel Stanley, and Vikram Adve. Program Control Language: A Programming Language for Adaptive Distributed Applications. *Journal of Parallel and Distributed Computing (to appear)*, October 2002. (accepted for publication).

[18] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN '98 conference on Programming language design and implementation*, pages 85–96. ACM Press, 1998.

[19] Mary F. Fernández. Simple and effective link-time optimization of Modula-3 programs. *ACM SIGPLAN Notices*, 30(6):103–115, 1995.

[20] David Gay and Alex Aiken. Language support for regions. In *Proc. SIGPLAN '01 Conf. on Programming Language Design and Implementation*, pages 70–80, Snowbird, UT, June 2001.

[21] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, MA, 1983.

[22] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, Menlo Park, CA, 1996.

[23] Rajiv Gupta, Eduard Mehofer, and Youtao Zhang. Profile guided compiler optimizations. In *The Compiler Design Handbook: Optimizations and Machine Code Generation.* CRC Press, 2002.

[24] T. Halfhill. Transmeta breaks x86 low-power barrier, 2000.

[25] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *ACM SIGPLAN — SIGSOFT workshop on on Program analysis for software tools and engineering*, pages 54–61. ACM Press, 2001.

[26] Simon Peyton Jones, Norman Ramsey, and Fermin Reig. C–: a portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming*, 1999.

[27] Sumant Kowshik, Dinakar Dhurjati, and Vikram Adve. Ensuring Code Safety Without Runtime Checks for Real-Time Control Systems. In *Proc. Int'l Conf. on Compilers, Architecture and Synthesis for Embedded Systems (CASES02)*, Grenoble, France, October 2002.

[28] Chris Lattner and Vikram Adve. Data structure analysis: An efficient context-sensitive heap analysis,with applications. Submitted to the ACM SIGPLAN Conference on Programming Language Design and Implementation (2003).

[29] Chris Lattner and Vikram Adve. The LLVM reference manual: `http://llvm.cs.uiuc.edu/docs/LangRef.html`.

[30] Chris Lattner and Vikram Adve. Automatic Pool Allocation for Disjoint Data Structures. In *Proc. ACM SIGPLAN Workshop on Memory System Performance*, Berlin, Germany, June 2002.

[31] Chris Lattner et al. The LLVM web site: `http://llvm.cs.uiuc.edu`.

[32] Microsoft. The .NET Common Language Runtime, see web site at: `http://msdn.microsoft.com/net`.

[33] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[34] R. Muth. Alto: A platform for object code modification, 1999.

[35] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen. Instrumentation and optimization of Win32/Intel executables using etch, August 1997.

[36] Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 47–56. ACM Press, 2000.

[37] SGI. SGI Open64 Compiler, `http://open64.sourceforge.net/`.

[38] Zhong Shao, Christopher League, and Stefan Monnier. Implementing Typed Intermediate Languages. In *International Conference on Functional Programming*, pages 313–323, 1998.

[39] Michael D. Smith. Overcoming the challenges to feedback-directed optimization (keynote talk). In *Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, pages 1–11. ACM Press, 2000.

[40] Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, December 1992.

[41] T.B. Steel. Uncol: The myth and the fact. *Annual Review in Automated Programming 2*, 1961.

[42] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[43] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, pages 132(2):109–176, February 1997.

[44] David Ungar and Randall B. Smith. Self: The power of simplicity. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 22, pages 227–242, New York, NY, 1987. ACM Press.

[45] Frederic Vivien and Martin Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation*, pages 35–46. ACM Press, 2001.

[46] Robert Wilson. An overview of the SUIF compiler system. Unpublished manuscript, Stanford University.

[47] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer-Ann M. Anderson, Steven W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *SIGPLAN Notices*, 29(12):31–37, 1994.