# Low Level Virtual Machine: A Framework for Link-time and Runtime Optimization

Chris Lattner
University of Illinois at Urbana-Champaign
lattner@cs.uiuc.edu

Vikram Adve
University of Illinois at Urbana-Champaign
vadve@cs.uiuc.edu

## Abstract

This paper introduces a new typed assembly language we call a Low Level Virtual Machine (LLVM) aimed primarily at better analysis and optimization of modern applications. Trends towards increased componentization, modularization, and object-oriented designs increase the importance of interprocedural optimizations, link time optimizations, and runtime optimizations. The LLVM approach enhances the effectiveness of all three optimization approaches. In particular, it provides the ability to perform high-level, language-independent, interprocedural optimizations at link time, without exporting a compiler internal representation or object code annotations. It also enables high-level optimization at runtime because its design and compilation architecture ensure that there is a simple mapping between "final" optimized LLVM code and generated native code. In this paper, we describe the LLVM language and compilation strategy and give examples of some novel optimization capabilities that are possible using this representation.

## 1. INTRODUCTION

Modern programming languages (e.g., C++, Java, and Csharp) and software practices (e.g., component libraries, dynamically shared libraries, and portable bytecode) aim to support more reliable, flexible and powerful software applications. Unfortunately, these languages and software practices are poorly supported by current compilation strategies, which are predominantly static in their approach, resulting in very significant performance penalties for modern software design. For example, object-oriented languages substantially complicate static analysis because of the widespread use of pointers, dynamic method dispatch, and numerous small methods. Component libraries and dynamically linked shared libraries are typically not addressed by static optimization at all.

Broadly, there are three major approaches to addressing these challenges, and a combination of all three is likely to be important: (a) source-level interprocedural optimization, (b) link-time interprocedural optimization, and (c) runtime optimization. Source-level interprocedural optimization can bring powerful optimization techniques to bear, but important parts of an application (particularly library code) are often not available to the source-level optimizer. In practice, this approach has also proven difficult to use for large applications because it can require significant changes in the development process (e.g., to the inputs for *make* to make complete or nearly complete application source code available. For both these reasons, several commercial compilers now attempt to perform interprocedural optimizations at link-time [2, 8].

The other two choices — link-time and runtime optimization — share one significant limitation in the traditional model of compilation. Specifically, traditional source-level compilers generate machine-level object code, which can be very difficult to analyze and transform [23]. Link-time and runtime optimizers working with object code [3, 9, 17, 23, 27, 7, 25] are generally limited to low-level optimizations such as profile-driven code layout optimization, interprocedural register allocation, constant propagation, and dead-code elimination. Several commercial and research systems for link-time optimization alleviate this problem by communicating information from the static compiler to the link-time optimizers, either by *exporting an internal representation (IR) of the program* [11, 2, 8] or by adding special annotations to the object files [29]. This approach is difficult to use with code generated by unrelated third-party compilers, unless the exported IR or annotations are standardized. It also does not address the problem for runtime optimizers, since the static IR would probably be too large to be of practical use.

Runtime optimization has a second key limitation, namely its runtime overhead, which restricts how much effort can be expended on optimization. Program analysis is often a dominant cost in compiler optimizations, and reducing the need for runtime analysis is a fundamental challenge that must be addressed for more effective runtime optimization.

In this paper, we present a compilation strategy that directly addresses the challenge of optimizing object code at *both* link time and runtime, and does so *without* exporting compiler-specific internal representations or adding annotations to object code. It can also reduce the runtime cost of program analysis by making significant analysis information available to the runtime optimizer.

In particular, we present a Low Level Virtual Machine (LLVM) – an instruction set that uses primarily low-level operations, similar to a RISC assembly language, but provides rich information about the operands. First, it includes

high-level but *language-independent* type information such as pointers, structures and arrays, along with standard operations on these types. All operations in LLVM are typed, but an explicit cast instruction allows languages like C to be implemented. (In this respect, LLVM differs from Typed Assembly Language [21], because our primary goal is performance.) In fact, the cast instruction exposes exactly where potential type violations might occur. This enables optimizers to exploit type information where available, without being unduly penalized by occasional type violations. Second, the representation is in static single-assignment (SSA) form, which can enable many efficient optimization algorithms without the expense of computing SSA at link-time or runtime.

Arbitrary source-level compilers can generate LLVM object code, after performing any optimizations they choose. For example, with modest effort, we have added an LLVM back end to GCC, which generates LLVM code after performing machine-independent optimization. Our results show that the LLVM representation is also significantly more compact than native machine code. A linker links the LLVM object code files of an application, performs link-time optimizations, and then generates native machine code for the target architecture. The code generation phase also preserves a detailed mapping between the "final" LLVM code and the native machine code, both at basic block and instruction level. The design of LLVM facilitates a fairly precise mapping, as explained in Section 4. We have implemented a link-time optimizer and code generator for the SPARC v9 architecture that retains such a detailed mapping. (Evaluating the accuracy, benefits, and costs of this mapping information for runtime optimization are outside the scope of this paper.)

There are two key differences between LLVM and other virtual machines such as the Java VM [19], Microsoft's Common Language Runtime (CLR) [20], and similar research systems [28, 1]. First, LLVM is a much lower-level representation, and does not include constructs like classes, inheritence, virtual functions, and garbage collection. In fact, we expect that LLVM would be used when *compiling the virtual machines* for these systems (e.g., the non-Java portions of a JVM and its runtime library). Second, some of the major goals of these other systems are bytecode portability and code safety, whereas the primary goal of LLVM is better link-time and runtime optimization (for any given architecture). In fact, our proposed compilation strategy for LLVM *does not include a Just-In-Time (JIT) compiler*; the code-generation is performed at link-time.

The LLVM approach makes some interesting and powerful optimizations possible at link-time. One example is data structure reorganization for structures and arrays [6]. A second is to eliminate unused exception handling code generated by C++ compilers, which can occupy significant space in executable images [26]. Both these are fundamentally interprocedural problems that require extensive high-level information about data accesses and control flow. With LLVM, these optimizations are now possible *at link-time*. LLVM also makes it possible to perform offline optimization *in the field*, using (for example), the system architecture described by Kistler and Franz.

To summarize, LLVM supports a strategy of compilation that distributes the optimization effort between static, link-time, and runtime compilers. In particular, the LLVM approach provides 4 capabilities:

- The ability to work with multiple source-level compilers, and allow them to perform arbitrary static optimization.
- High-level, language-independent, global and interprocedural optimizations at link time.
- High-level, language-independent runtime optimizations, by preserving the mapping between LLVM and native code. Note that LLVM is specifically designed to enable such mapping (by using low-level operations).
- The ability to perform offline optimization transparently in the field, after code installation.

To our knowledge, no other current system provides all these capabilities. The closest is perhaps Microsoft's Common Language Runtime, which is designed to enable both link-time and runtime optimization in a language-independent manner, but significantly limits the transformations performed by static source-level compilers. It is also constrained by the need to provide stringent safety, portability, and memory management guarantees, as with JVM bytecode. (We emphasize that LLVM is not designed to be an alternative to CLR or JVM but rather a lower-level compilation platform *for a given system*. We also expect that our research on link-time and runtime optimizations using LLVM will also benefit CLR and JVM.)

The next section describes the design and implementation of LLVM. The following two sections describe our strategies for interprocedural link-time optimizations, and for runtime optimization. We then describe some preliminary results on a previously proposed interprocedural optimization (structure splitting [6]), which requires substantial high-level information but can be performed at link-time with LLVM. Next, some novel long-term uses of the LLVM platform will be considered. Finally, we compare our approach with related work, and conclude with a brief summary of the direction of our near-term work.

## 2. LLVM DESIGN

We designed LLVM to be a low-level, high performance, platform neutral, virtual machine (VM). The design of LLVM reflects our goals of building a high performance post-linktime compilation system, and addresses the need for an expressive VM capable of representing programs from any source language. LLVM is a typed assembly language[citation] that uses Static Single Assignment form as its base representation and RISC-like three-address code for most of its instructions (add, sub, mul, shifts, etc...). The figure below shows a fragment of C code and the generated LLVM code, including the `phi` instruction, used to represent SSA $\phi$ nodes.

SSA representation provides a simple and natural way of representing use-def edges in the program, and is the basis for many common sparse optimizations. Although SSA, coupled with three-address code, is a strong platform for many low-level optimizations, it is inadequate for high-level transformations and analysis. Because these applications are very important for our overall strategy, we designed a strong typing system for LLVM and four unconventional instructions that are critical to type safety analysis (`getelementptr`, `malloc`, `alloca`, and `free`).

```
/* C Source Code */
int SumArray(int Array[], unsigned Num) {
 unsigned i; int sum;
  for (i = 0, sum = 0; i < Num; ++i)
    sum += Array[i];
  return sum;
}

; LLVM assembly code
int "SumArray"([int]* %Array, uint %Num)
begin
        %c0   = setge uint 0, %Num
        br bool %cond62, label %bb3, label %bb2
bb2:    %sum1 = phi int [%sum2,%bb2], [0, %bb1]
        %i1   = phi uint [%reg119,%bb2], [0, %bb1]
        %elem = load [int]* %Array, uint %i1
        %sum2 = add int %sum1, %elem
        %i2   = add uint %i1, 1
        %c1   = setlt uint %i2, %Num
        br bool %cond20, label %bb2, label %bb3
bb3:    %sum3 = phi int [ %sum2, %bb2 ], [ 0, %bb1 ]
        ret int %sum3
end
```

**Figure 1: LLVM code for C fragment**

## 2.1 LLVM Type System

The type system in LLVM is a simple constructive type system with several primitive types (`void`, `bool`, signed and unsigned integers from 8 to 64 bits, `float`, `double`, `label`, `opaque`) and 4 derived types (method, structure, array, pointer). Each SSA value is typed, and individual instructions have strict rules about the possible types of their operands. For example, the two operands to the `add` instruction must have the same type, and the first operand to the `load` instruction must be a pointer type.

In order to support non-typesafe languages, LLVM includes a `cast` instruction that is used to convert a value from one type to another. Because LLVM requires types to be consistent, some casts are required that do not affect program data. For example, loading float value from an `int*` value requires casting the value to a `float*` type before the load, even though no reinterpretation of data occurs.

## 2.2 Unconventional Operators

In order to fully exploit the type system, LLVM includes instructions that directly represent memory access idioms, which makes LLVM a particularly expressive language. These instructions make it possible to eliminate casts from the source program, making the program smaller and more type-safe. Indeed, for a type-safe input program (which is a property independent of the input language), it is possible to eliminate all casts from the LLVM source that do not reinterpret data.

The `getelementptr` instruction is used to transform a pointer to a value of a structure or array type into a pointer to an element of the value. This allows standard addressing to be represented in LLVM without explicit pointer arithmetic. Figure 2 shows three ways to load the `int` element of the allocated structure. Approach A completes this task using standard pointer arithmetic. Notice that this solution depends on the data layout of the target machine. Approach B uses `getelementptr` as a solution, and Approach C uses the indexed form of the `load` instruction. Because the fused form is available, typically the only time an explicit `getelementptr` instruction is found is when the source program takes the address of a field (e.g. to pass the pointer into a function call).

Note that although it is possible to represent these operations without the `getelementptr` instruction, doing so makes the program more verbose and difficult to analyze. The `getelementptr` instruction provides a compact way to describe why the derived new pointer has a particular type. Multiple levels of indexing in the same instruction also provide a compact serialized representation.

Memory management in LLVM can be directly represented through the `malloc`, `free`, and `alloca` instructions. These instructions are used to allocate heap memory, free heap memory, and allocate stack memory, respectively. The sole purpose of these instructions is to provide a type-safe way of handling memory, similar to the C++ new and delete operators (without the constructor or destructor calls). In contrast to this approach, the C style `malloc` function call returns an untyped pointer that is difficult to analyze. The `malloc` and `alloca` instructions take a type to allocate and, optionally, the number of elements to allocate. The `free` instruction takes a pointer value to release.

The `alloca` instruction allocates stack memory of a particular type, and is used as the primary means to allocate local variables that must occupy a memory location. In C, most automatic variables may be safely promoted to SSA values, but some may not (e.g. their address was taken in the source program). Because these values must have an address, they are allocated on the stack, which provides automatic memory reclamation on function return. One interesting side effect of this strategy is that LLVM contains no "address of" operator. All addressable entities are iden-

```
; Stack allocate a structure of 3 elements
%aStruct = alloca { [10 x float], int, uint* }

; Approach A:  Use standard pointer arithmetic
%t1 = cast uint 40 to int*
%t2 = cast { [10 x float], int, uint* }* %aStruct to int*
%intPtr = add int* %t1, %t2
%intVal = load int* %intPtr

; Approach B: Use the getelementptr instruction
%intPtr = getelementptr { [10 x float], int, uint* }* %aStruct, ubyte 1
%intVal = load int* %intPtr

; Approach C: Use the fused getelementptr-load form to perform the load directly
%intVal = load { [10 x float], int, uint* }* %aStruct, ubyte 1
```

**Figure 2: Three approaches to loading a field from a structure**

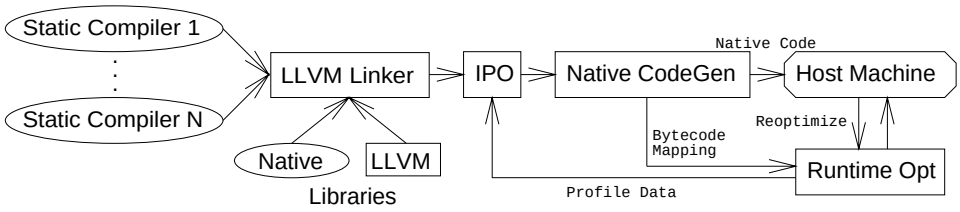


**Figure 3: LLVM system architecture and compilation process**

tified by their address, and addressable memory locations are explicitly allocated.

## 2.3 Function calls and exceptions

LLVM provides two function call instructions, which abstract out the calling conventions of the underlying machine and simplify program analysis. The simple `call` instruction takes a pointer to a method to call (which is a global constant for direct calls), as well as the arguments to pass in (which are passed by value). The `invoke` instruction is used for languages with destructors or `catch` blocks, in order to implement exception handling.

LLVM implements a stack unwinding[cite A Single Intermediate Language That Supports Multiple Implementations of Exceptions, PLDI2000] mechanism for "zero cost" exception handling. A "zero cost" exception-handling model indicates that the presence of exception handling causes no extra instructions to be executed by the program if no exceptions are thrown. When 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 is used to invoke handlers when 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 method pointer and method 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.

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 languages exception handling semantics. In this representation, exception edges are directly specified and visible to the LLVM framework, preventing unnecessary pessimization of optimizations when exceptions are possible. This representation is also conducive to exception handling optimizations discussed later.

## 2.4 System Architecture

The LLVM system is a design for a balanced compilation environment, that focuses on high performance. To support this, we use LLVM bytecode as the common code representation for all phases of our compilation strategy [Figure 3]. Our design permits retargeting multiple conventional static compilers to LLVM bytecode and the interprocedural backend. This allows us to use the existing front-ends and the powerful global optimizers in these compilers for our work.

After linking, the resultant LLVM bytecode is interprocedurally optimized, with profile data from previous executions of the program. When optimization is complete, native code generation compiles the LLVM bytecode into a native executable, which is directly executable by the user on the host system.

As an optional component, the LLVM runtime optimizer (see Section 4 ) can be used to fine-tune the application as it runs. If enabled, simple profiling instructions are inserted into the generated machine code and mapping information is generated. This mapping information is used to correlate the native profiling results back to the LLVM bytecode representation.

With this setup, the runtime optimizer can be used to tune application performance to varying runtime conditions and settings without having to dynamically compile the entire application from scratch when execution starts. In addition to saving compilation time, this approach also enables the operating system to share code pages between multiple in-

stances of the program, something that most dynamic optimizers do not allow. The runtime optimizer performs translation on the LLVM representation of a function, generating code as neccesary in response to application behavior.

## 2.5 Retargeting an existing static compiler

Retargeting a preexisting compiler for a RISC platform to the LLVM architecture is a tractable job. LLVM provides a simple uniform instruction set, without the complexities of many RISC architectures. It does not require register allocation or instruction selection, and can often be derived directly from the compiler's internal representation.

For our initial implementation, we designed a back end for GCC (the Gnu Compiler Collection). When we started, GCC provided experimental support for transforming RTL (Register Transfer Language, the GCC IR) into SSA form. We based our code generation pass on this RTL-SSA form, building LLVM code directly from this low-level representation.

As a very low-level intermediate representation, RTL lacks data type information except for instruction operand size. For LLVM, we were required to reverse engineer the LLVM type information from the RTL. We were able to configure the LLVM GCC machine description file to treat pointers as data size distinct from other integer types, which provided a minimal classification of value types. In addition to this classification, we used function prototypes (obtained from the GCC abstract syntax tree), and type information implicit in instructions (e.g. signed vs. unsigned division) to get basic type information about the program.

From the legal, but poorly typed, LLVM output program, we apply a series of transformations to raise the types in the program, using the high level types that stem from the function prototypes. This transformation is front-end independent, allowing it to be used by many different future front ends without modification (because it is an LLVM to LLVM transformation). Unfortunately, the algorithm used is outside the scope of this paper.

# 3. LINK-TIME OPTIMIZATIONS

An important aspect of LLVM is that it supports high level transformations at link-time, a feature which has traditionally only been offered by source level interprocedural optimizers. Interprocedural optimization (IPO) is an increasingly important phase of a modern compiler, due to the increasing abstractions and decoupling of application implementations. Link-time is a good point in the compilation system to perform IPO, because most (if not all) of the application code is available for analysis and transformation. In particular, many transformations require the entire application to be transformed at the same time for correctness.

Classical interprocedural dataflow optimizations like constant propagation and dead code elimination can be directly implemented in LLVM, taking advantage of the sparse SSA forms of the algorithms. Additionally, LLVM allows more aggressive optimizations than are typically performed in an interprocedural optimizer, such as dead method elimination and dead global elimination. This section describes several optimizations that are enabled or simplified by the LLVM representation.

## 3.1 Structure layout reorganization

Many transformations have been proposed for reorganizing data structures to improve cache behavior, including loop and structure element transformations. One optimization enabled by LLVM is structure splitting [6]. Structure splitting divides a structure into a hot and cold portion, in an attempt to make the hot portion fit better on a cache line, and pack into memory better. The cold portion of the structure is split off into a secondary, dynamically allocated portion, which is no longer adjacent to the original structure.

While access to the cold portion of the structure suffers from an extra pointer indirection to access it, the hot portions of the structure are smaller and can be packed better into the cache lines of the host machine. This transformation is an intrinsically interprocedural transformation, and the entire program must be available for the transformation (otherwise, some portions of the program assume one layout, and other portions of the program assume another).

One interesting aspect of using LLVM, however, is that we are able to apply splitting to programming languages that can depend on the layout of data. Because LLVM does not require programs to be type-safe (and many C programs are not), an important portion of our implementation is the analysis to prove the transformation safe.

## 3.2 Structure Field Access Analysis

In general, it is not safe to transform the memory layout of data structures in C programs. Because of this, most commercial compilers do not attempt to apply memory-reordering transformations to C programs. Despite current practice, we hypothesize that most C programs, in an effort to remain portable, do not depend on the memory layout of their data structures. Programs that are dependant on the memory layout of their data structures typically have small target dependant portions, separated from the main logic of the application.

As described earlier, when a program is compiled into LLVM its representation is raised to use `getelementptr` instructions instead of explicit addressing where possible. This transformation is capable of eliminating all pointer-to-pointer casts in a type-safe program, and is specifically designed to decouple the semantics of the source program from the actual memory accesses it performs. We perform structure field access analysis after the program has been raised.

Structure field acccess analysis is a flow insensitive interprocedural analysis that inspects pointer cast instructions in the source program. The following pseudo-code summarizes how we determine whether a structure type is not safe to transform:

```
let UnsafeTypes = ∅
∀ instruction I ∈ Program
  if not TypeSafeInstruction(I)
    ∀ operand o ∈ I
      if Type(o) is a pointer type
        UnsafeTypes = UnsafeTypes ⋃ { Type(o) }

∀ type t ∈ UnsafeTypes
  ∀ type t2 ∈ SubTypes(t)
    UnsafeTypes = UnsafeTypes ⋃ { t2 }
```

This algorithm operates by scanning the program for instructions that are not considered "type-safe" when applied

to pointers. Only instructions that use pointer value in a type-safe, memory layout agnostic manner (`alloca`, `malloc`, `free`, `load`, `store`, `getelementptr`), or delay use of the pointer until a later point in time (`phi`, `call`, `invoke`) are considered typesafe. All other instructions cause the operand types to be added to the unsafe set (for example, `add`'ing two pointers together exposes their memory layout, which marks the types as unsafe to transform).

After the initial set of unsafe types is determined, a closure over the contained types must be determined. LLVM types form potentially cyclic graph structures, where every node in the graph is a "subtype" of the original type. As a constructive type system, derived types build on other more primitive types, defining edges in the graph. Figure 3.2 shows a graphical representation of binary tree type from the `treeadd` benchmark. The *SubTypes* function returns the set of nodes (types) in the graph for a given type.

The first pass of the algorithm runs in O($i$) time, where $i$ is proportional to the the number of instructions in the program. Using worklists, the second pass of the algorithm runs in O($t$) time, where $t$ is proportional to the number of types in the final set, and is strictly less than $i$. Therefore, the algorithm is linear in the size of the input program.

As shown in Section 5.2, this very simple analysis is powerful enough to prove that many programs are type-safe, enabling the use of advanced memory reorganizing algorithms typically used by type-safe high level languages.

## 3.3 Optimization of Exception Handling

Exception handling is an important part of modern programming languages, and many techniques for efficient handling of exceptions have been devised. As mentioned before, LLVM provides infrastructure to implement a 'zero cost' exception handling model. However, even with a zero cost model for exception handling, the possibility of an exception occurring increases the static size of the program executable (for exception handling tables), and optimization opportunities may be lost due to the extra control flow edges in the program.

Prior work eliminating exception handling [26] has shown that much of the cost of exception handling can be eliminated. Schilling's work is based on an interprocedural AST level optimization. We propose extending Schilling's algorithm to be a linktime interprocedural optimization, instead of a source level optimization. At this time, significantly more opportunities to eliminate exceptions can be found by performing the optimization during the post-link optimization phase of compilation. At this phase, the benefits of the LLVM exception representation can be exploited.

In C++ and Java [14], functions without a try block, destructible local object, or exception specification (note that Java monitors are effectively destructible objects, and that exception specifications are statically enforced) have no cleanup action in the case of an exception. Because of this, calls within the function can be safely converted to the LLVM `call` instruction instead of using the `invoke` instruction, eliminating the table entry for the call. In LLVM, this is the straightforward way that exception-handling overhead is eliminated from a function.

Extending this, a simple traversal of the call graph for an input program can determine whether or not a function calls (directly or through other functions) the LLVM standard library function to access exception-handling data. If a function can not call this function, the data is dead, and the `invoke` can be converted to a `call` instruction, eliminating the 'hidden' overhead of the exception handler. Cleanup handlers that correspond to the invoke instruction may now be dead, further increasing optimizations possibilities.

# 4. RUNTIME OPTIMIZATION OF NATIVE CODE

LLVM is designed to assist runtime optimization, in addition to link-time. Furthermore, unlike other virtual machine based systems such as those for SmallTalk, Self, Java, or Microsoft CLR, the goal in LLVM is to do native code-generation at link-time, and only do optimization on key parts of the native code at runtime.

The key to providing these capabilities is to maintain a detailed mapping between the final form of LLVM code and the generated native code. The precision of this mapping may affect the granularity at which runtime optimizations will be most effective.

We aim to use two different granularities for runtime optimization: traces and methods. Trace-based optimization focuses optimizations on frequently executed dynamic sequences of instructions called traces [12, 4]. The traces being gathered in our system only include one or more complete basic blocks. Method-based optimization is widely used in many adaptive JIT optimizers [16, 5, 28]. This focuses optimization effort on entire procedures that are frequently executed (or that consume a large fraction of execution time [**?**]).

## 4.1 Mapping Between LLVM and Native Code

The link-time optimizer performs one or more transformations on the LLVM code for an application after it is linked, as shown in Figure 3. The "final" LLVM code resulting from these transformations is input to the link-time code generator, which also generates LLVM-to-native-code maps.

The quality of the mapping information produced by the code generator can be reduced by two factors:

1. First, LLVM-to-LLVM transformations that replace high-level operations with low-level ones will reduce the semantic information available in the "final" LLVM code.

2. Transformations performed by the code generator after generating native machine instructions could reduce the precision with which LLVM instructions (or sequences) can be mapped to native code instruction sequences.

LLVM is designed to minimize both types of information loss. First, and most important, the low-level instructions of LLVM ensure that very few instructions are generated for each LLVM instruction, with most register to register operations mapping one-to-one. The longest sequences are generally 5 to 8 SPARC v9 instructions (typically involving loading 64-bit constants into a register). Second, structure field references directly translate into simple base-offset addressing, so that such references rarely need to be optimized using redundancy elimination techniques. Third, type information for values should not be significantly affected by LLVM transformations, including both primitive values and structure values. The one case where significant information

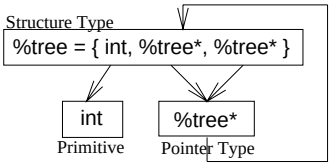


**Figure 4: Type graph for structure type in `treeadd`**

can be lost via LLVM transformations is for array operations, which must be lowered to explicit address arithmetic so that the address computations can be optimized via induction variable substitution and partial redundancy elimination. Nevertheless, simple annotations on the LLVM load instructions could preserve this information.

The link-time code generator uses 4 major passes: instruction selection, instruction scheduling, global register allocation, and peephole optimization. An additional pass of instruction scheduling after register allocation can also be useful for methods that incur significant spill code.

The instruction selector computes an initial one-to-many mapping of LLVM instructions to machine instructions. Our current instruction scheduler is a basic-block list scheduler that can reorder instructions within basic blocks but does not move instructions across basic blocks. With such a scheduler, *each basic block of LLVM code maps to a single basic block of native code*. Furthermore, the instruction level maps are still available, but the native instructions for a particular LLVM instruction may be reordered with respect to other native instructions in the block. The later phases (register allocation and peephole optimization) also do not move code across basic block boundaries, and therefore do not significantly degrade the precision of the mapping.

The major potential source of degradation during code generation would be from more aggressive schedulers such as software pipelining techniques, which reorder code across basic blocks. We believe that software pipelining could be performed *on LLVM code* directly, *before generating native code* – this would eliminate this problem directly. The major challenge is computing an initiation interval for a loop, based on the cost of its instructions [22]. Because the mapping between LLVM and native instructions is very simple, it appears feasible to estimate the approximate final cost of the native code from the LLVM code for a loop, which would make this approach feasible. Alternatively, we can restrict runtime optimization to work with methods instead of traces, so that entire methods are optimized at runtime using the LLVM code for the method.

Perhaps most importantly, the above transformations on the native code will not invalidate the mapping between machine instruction *operands* and final LLVM operands, which is computed by the register allocator. Since most high-level information in LLVM is associated with the operands (include type information and SSA def-use edges), this high-level information should be directly available for the runtime optimizer.

## 4.2 Optimization Strategies

Given the mapping information described above, we can use two different strategies for exploiting this mapping information at runtime (these are orthogonal to the choice of granularity). We describe these for trace-based optimization, and then briefly touch on method-level optimization.

We are currently implementing a trace-collection system for LLVM, using a combination of previously published techniques [4]. In particular, unlike the pure runtime systems such as Dynamo, LLVM makes it feasible for us to use significant static analysis to reduce the overhead or improve the quality of runtime traces. For example, loop nests with loop-invariant bounds and no internal branches do not need any explicit tracing at all, but merely a test to compare the loop count against the threshold used to choose "hot" traces.

One optimization approach, widely used by JIT compilers, is to regenerate optimized native code from bytecode for a hot trace or method. The disadvantage of this approach is that it requires a complete code-generation pass at runtime, which precludes the use of aggressive code generation techniques.

Our second, and preferred, approach is to optimize traces of native code directly, using the LLVM code purely as a source of high-level information. The advantage is not having to re-generate native code. The major disadvantage is that a sequence of transformations on native code will not all be able to use LLVM as a source of information because successive transformations will make the original mapping obsolete.

Both these approaches can also be applied to method-level optimization. However, if method-level is only used when basic-block level mapping information is invalidated, then the only way to exploit LLVM information may be to regenerate (optimized) native code from the LLVM object code at runtime. This is likely to be worthwhile only if significant improvements in code quality are possible using runtime information, such as runtime constants. s

# 5. PRELIMINARY RESULTS

We have implemented most of the static portion of the LLVM system, including a backend for GCC that generates LLVM code, the interprocedural structure access analysis and transformation pass, a code generator for 64-bit Sparc Solaris platforms and a large amount of useful infrastructure for our work. Infrastructure includes a set of bytecode tools for assembling and dissassembling LLVM code, a set of standard global optimizations, and a level raising framework. To assist with development and guide optimizations, we have a very simple interpreter/debugger/profiler for use with LLVM bytecode applications.

The level raise transformation is currently used to convert code produced by GCC to "higher" level forms. This process eliminates machine specific constructs that impede analysis and prevents transformations, as well as reducing the amount of nontrivial code present in the GCC code generator itself. To verify that our transformations are correct, we compared output from the Sparc backend, the native platform compiler, and the LLVM interpreter.

In this section, we evaluate the effectiveness of our structure field access analysis and structure reordering transfor-

| Benchmark | LOC | Bytecode Size | Stripped Native |
|---|---|---|---|
| em3d | 906 | 6516 | 14112 |
| health | 728 | 4552 | 11448 |
| mst | 593 | 4664 | 10376 |
| perimeter | 478 | 3668 | 8872 |
| power | 816 | 7032 | 12992 |
| treeadd | 266 | 1028 | 6224 |
| tsp | 583 | 4200 | 10760 |

**Table 1: Benchmark Properties**

mations. We use the C version of the Olden benchmark suite as the basis for our work, because it is well studied and memory intensive. Table 1 lists the component benchmarks of the Olden suite and their characteristics.

Due to minor bugs in our GCC backend, we are unable to process the bh, bisort, and voroni benchmarks, so they have been omitted (we expect these issues to be fixed well before the final submission date).

## 5.1 Evaluation of Bytecode and Backend

Table 1 lists the size of the compiled bytecode each application, along with the size of the native code generated by the platform C compiler. LLVM code is consistently smaller than the native machine code for the benchmarks, ranging from 17-54% of the stripped native executable size. Because we intend to include LLVM bytecode with the executable program itself, it is important for it to be compact. Although we have not attempted to optimize bytecode encoding for size yet, we expect that applying well studied techniques [13] will yield good results.

The LLVM Sparc backend, although immature, is showing great promise on pointer based codes such as the olden benchmarks. For example, on the `treeadd` benchmark, we consistently match the performance of the Sparc platform compiler when run with agressive optimizations turned on (`-xO5 -xarch=v9`). This result is a combination of LLVM building on the strengths of the GCC scalar optimizer, coupled with the weakness of traditional compiler approaches to memory based codes.

## 5.2 Structure Field Access Analysis

We have implemented the interprocedural flow insensitive structure field access analysis presented in Section 3.2. This analysis is able to successfully prove that the structure types are reorderable in the `treeadd`, `perimeter`, `health`, `power`, and `tsp` benchmarks. The `em3d` benchmark fails analysis due to an unimplemented feature in our level raising framework (induction variable analysis).

The `mst` benchmark is interesting because it shows an important failure mode of the structure field access analysis. `mst` uses a "generic" hash table implementation in C, where entries in the hash table are stored as void pointers. Because the main structure type of `mst` is cast to and from the C `void*` type, structure field access analysis conservatively assumes that the data could be modified in an unknown way. For this reason, data reordering transformations are inhibited on the affected structure types.

Data structures and algorithms that operate on `void*` data are fairly common in C programs, due to the lack of generics support in the language (Even the standard C library `qsort` function uses a variant of this technique).

One straightforward, but untested, approach to handling this scenario is to track values as they are propogated through the generic code in question. Since the hash table code is itself typesafe, we can track loads and stores into the different fields of the hash table data structure. Operating in an interprocedural framework allows us to directly inspect the flow of values through the library, tracking whether or not they are cast back to the orignal type, some other type, or modified in the library.

## 5.3 Structure Reordering Transformations

We have implemented several structure reordering transformations, including general field permutation support and field elimination. Structure field access analysis is sufficient to prove that fields of structures are dead and can be eliminated by statically analyzing uses of a field. As with scalar variables, structure fields are dead if there are no uses/loads of the field. The Olden benchmarks have several fields that are assigned to but never read. Eliminating fields is a straightforward way to improve cache performance through reduced memory consumption.

Because of the limitations of our native code generator backend (minor bugs to be ironed out), the range of experiments was unnaturally restricted to simple test cases. We adapted a version of the `treeadd` benchmark (which constructs a simple binary tree, then traverses it 100 times, adding up the data fields), to include large dead fields mixed in among the child pointers. While extreme, this test case provides a useful limit analysis for what we can achieve. We converted the main data structure to:

```
struct tree {
  int val;             int pad1[64];
  struct tree *left;   int pad2[64];
  struct tree *right; int pad3[64];
}
```

With this testcase, we tested the result of several different transformations to the modified treeadd program. Table 2 shows the results of these different transformations. The first two columns correspond to the modified treeadd benchmark as compiled by the LLVM backend and the platform native compiler with strong optimizations turned on. Because the performance is dictated primarily by the data layout, both compilers are very close in execution time.

The third column of the table are execution times for when a transformation that sorts structure elements by their data size is enabled. The transformed C type becomes:

```
struct tree {
  int val;
  struct tree *left, *right;
  int pad1[64], pad2[64], pad3[64];
}
```

Because this transformation allows access to all of the pertinent data of the tree node with only a single cache line fetch, performance is dramatically improved. The fourth column is the same program with dead field elimination turned on. Because there are no reads of the values from the fields, the fields are dead and can be eliminated. Doing so allows several different tree nodes to be fetched in the same cache line read, again improving performance substantially.

Although this is a somewhat contrived example, it does provide a useful limit analysis, showing that reducing cache

| llvm | native | sorted | dfe |
|---|---|---|---|
| 19.94s | 21.35s | 13.29s | 5.82s |

**Table 2: treeadd results**

line pollution and usage is important. Traditionally, compilers have not been very agressive with their treatment of memory references. We show that, similar to array transformations, structure transformations can have a large effect on overall application performance.

# 6. RELATED WORK

Virtual machines have been used widely for implementing safe, portable or dynamic languages like Smalltalk, Self and Java, and language-independent systems like Microsoft's Common Language Runtime. We contrasted the goals of our work with these previous systems in some detail in the introduction. To summarize that discussion, LLVM is a lower-level representation that could be used as a platform to optimize these other virtual machines (just like any other application written in a directly compiled language). LLVM's primary goal is high-performance compilation, not portability and bytecode safety.

A key question is whether the bytecode representations used for CLR or Java [19, 20, 1, 13] would not serve as a good basis for the goals of this work. In particular, CLR aims to support a wide-range of languages. As noted briefly in the introduction, the key limitation shared by all these representations including CLR is the need to provide stringent safety, portability, and memory management guarantees. For example, all these representations *define* powerful memory management and exception-handling semantics, which must be implemented in their runtime systems. In contrast, LLVM provides very low-level primitives for memory allocation and exception handling that can be used to implement higher-level semantics used by these or other languages.

The design of LLVM has some similarities to SafeTSA, a bytecode representation recently proposed as an alternative to Java bytecode [1], though LLVM was developed concurrently. In particular, SafeTSA is strongly typed and also based on an SSA representation, but still enforces safety semantics on input programs, much as other bytecode representations do.

LLVM is essentially a rich assembly language, similar to Typed Assembly Language (TAL) [21]. TAL is a strongly-typed RISC assembly language designed to serve as a target for optimizing compilers for type-safe languages, allowing type information to be exploited throughout the optimization process. A key difference from LLVM is that TAL does not provide a mechanism to permit type violations, preventing, for example, arithmetic on pointer values. LLVM can be viewed as a compromise design that allows type information to be exploited for all but the native code generation phase of an optimizing compiler, and for the type-safe subsets of programs in both type-safe and non-type-safe languages. A design difference is our use of SSA form, which enables efficient, sparse algorithms for many optimizations. TAL also provides a powerful basis for certifying safety properties of assembly code from untrusted compilers, which is not possible in LLVM and not one of its goals.

Several systems have performed interprocedural optimization at link-time. One subset of these focus on optimizing assembly code for a given target machine without any additional information from static compilers [23, 27, 7, 25]. Such systems focus primarily on machine-dependent optimizations such as interprocedural register allocation, profile guided code layout, dead code elimination, plus some simple versions of machine-independent optimizations such as copy propagation and loop-invariant code motion [23]. Muth et al. provide an insightful discussion into the challenges of optimizing machine level code [23].

The remaining class of link-time optimizers rely on information from the static compiler (either in the form of an IR or annotations) [29, 11, 2, 8]. As noted in the introduction, this approach is difficult to apply for code generated by third-party compilers, unless the exported information is standardized. It is also an expensive approach to use for runtime optimization, except perhaps with relatively lightweight annotations. Since LLVM is an assembly language, it provides an implicit standard for all the information it contains (like any other assembly language). It is also quite compact, as noted earlier.

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 and offline optimization. Their system uses Slim Binaries [13] as its code representation, a very compact, compressed, tree-based code representation. Kistler and Franz observe that other representations could be used. LLVM could be directly used within this architecture, making it unnecessary to regenerate SSA form for every recompilation, as they suggest.

There have also been several systems that perform transparent runtime optimization of native code [3, 9, 17]. These systems inherit all the challenges of optimizing machine-level code [23] with the additional constraint of operating under very tight time constraints. In contrast, LLVM provides both type and SSA information, which serve as the basis for many optimizations. Even if instruction-level and basic-block-level mapping of LLVM to native code proves unsuccessful, method-level runtime optimization and code generation of frequently executed methods should be a practical strategy for runtime optimization. This would incur significanty less overhead and/or enable much higher-level optimizations than these previous systems.

Other runtime optimization approaches, including runtime code generation such as with VCODE [10, 24] and runtime code specialization such as in DyC [15], are designed for programmer-guided optimization, which is a fairly different goal than ours. The VCODE representation has some similarity in that it is used as a basis for runtime code generation, but its emphasis is on very fast code generation at a few cycles per instruction, and not as a basis for optimization (since its use is intended to be programmer guided).

A final body of related work is on specific transformations similar to the ones we describe in Sections 3 and 5. As mentioned in Section 3.3, Schilling shows [26] that exception-related overhead can be profitably eliminated from C++ code. We extend Schilling's work to operate at link time, and have reason to beleive that more exception overhead can be eliminated by using inter-translation unit information for analysis.

# 7. SUMMARY

This paper has presented a rich assembly language repre-

sentation, LLVM, designed to support more effective link-time interprocedural optimization, and runtime optimization. The paper also illustrates some of the novel optimization capabilities that are possible using this representation, including the ability to do interprocedural exception code elimination for code generated from C++, interprocedural data layout transformations at link time, and furthermore the ability to do such transformations on partially type-safe programs in unsafe languages like C.

One potentially valuable way to exploit such technology in practice is that any major processor architecture family would include a virtual instruction set (perhaps tailored to that architecture family), and a link-time and runtime optimizer based on the virtual instruction set would form part of the standard system software of a machine. Source-level compilers can choose to compile directly to native code, or to compile to the virtual machine instruction set in order to leverage the link-time and runtime optimizations. LLVM is an example of such a virtual instruction set that is designed to suit several different RISC processors, but currently not tailored to any particular processor family. Tailoring LLVM for the Intel IA-64 architecture is one goal of our future work.

LLVM makes possible some interesting and novel optimization capabilities, and exploring those capabilities are a key focus of our current work. We can also envisage some other novel research directions using LLVM as a platform. For example, we are exploring whether LLVM can be used as the basis for a virtual machine processor design [18, 17], where the hardware processor is completely hidden from user-level software, and all user-level software is implemented in a virtual instruction set that is emulated on the hardware using just-in-time code generation and adaptive optimization. Such an approach provides some very interesting flexibility for the processor designer [18, 17], but fundamentally depends on a virtual instruction set with the ability to do good runtime code generation and optimization. We believe that LLVM could be a very effective example of such an instruction set.

# 8. 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, 2000.

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

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. pages 1–12.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. pages 1–12.

[5] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeno dynamic optimizing compiler for java. In *Java Grande*, pages 129–141, 1999.

[6] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, 1999.

[7] R. Cohn, D. Goodwin, P. Lowney, and N. Rubin. Spike: An optimizer for alpha/nt executables, 1997.

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

[9] K. Ebcioglu and E. R. Altman. DAISY: Dynamic compilation for 100In *ISCA*, pages 26–37, 1997.

[10] D. Engler and V. retargetable. very fast dynamic code generation system, 1996.

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

[12] J. Fisher. Trace scheduling: A general technique for global microcode compaction, 1981.

[13] M. Franz and T. Kistler. Communications of the acm, 1997.

[14] J. Gosling et al. *The Java Language Specification*. GOTOP Information Inc., 5F, No.7, Lane 50, Sec.3 Nan Kang Road Taipei, Taiwan; Unit 1905, Metro Plaza Tower 2, No. 223 Hing Fong Road, Kwai Chung, N.T., Hong Kong.

[15] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. J. Eggers. An evaluation of staged run-time optimizations in dyc. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 293–304, 1999.

[16] D. Griswold. The java hotspot virtual machine architecture, 1998.

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

[18] S. S. J. E. Smith, T. Heil and T. Bezenek. Achieving high performance via co-designed virtual machines, 1999.

[19] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, 1997.

[20] Microsoft.

[21] G. Morrisett, D. Walker, K. Crary, and N. Glew. to typed assembly language, 1998.

[22] S. Muchnick. Advanced compiler design and implementation, 1997.

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

[24] M. Poletto, D. Engler, M. Kaashoek, t for, and F. Flexible. and high-level dynamic code generation, 1997.

[25] 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, 1997.

[26] J. L. Schilling. Optimizing away c++ exception handling. *ACM Sigplan Notices*, 33(8), Aug. 1998.

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

[28] D. Ungar and R. B. Smith. Self: The power of simplicity. In N. 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.

[29] D. Wall. Global register allocation at link-time. In *Proc. SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, 1986.