The LLVM Instruction Set and Compilation Strategy

Chris Lattner Vikram Adve

Computer Science Department Technical Report #UIUCDCS-R-2002-2292

University of Illinois at Urbana-Champaign

{lattner,vadve}@cs.uiuc.edu

Abstract

This document introduces the LLVM compiler infrastructure and instruction set, a simple approach that enables sophisticated code transformations at link time, runtime, and in the field. It is a pragmatic approach to compilation, interfering with programmers and tools as little as possible, while still retaining extensive high-level information from source-level compilers for later stages of an application's lifetime. We describe the LLVM instruction set, the design of the LLVM system, and some of its key components.

1 Introduction

Modern programming languages and software practices aim to support more reliable, flexible, and powerful software applications, increase programmer productivity, and provide higher level semantic information to the compiler. Unfortunately, traditional approaches to compilation either fail to extract sufficient performance from the program (by not using interprocedural analysis or profile information) or interfere with the build process substantially (by requiring build scripts to be modified for either profiling or interprocedural optimization). Furthermore, they do not support optimization either at runtime or after an application has been installed at an end-user's site, when the most relevant information about *actual* usage patterns would be available. The LLVM Compilation Strategy is designed to enable effective *multi-stage* optimization (at compile-time, link-time, runtime, and offline) and more effective profile-driven optimization, and to do so without changes to the traditional build process or programmer intervention.

LLVM (Low Level Virtual Machine) is a compilation strategy that uses a low-level virtual instruction set with rich type information as a common code representation for all phases of compilation. This representation allows us to preserve the traditional "compile-link-execute" model of compilation (allowing build scripts to be used unmodified), supports interprocedural optimization at link-time, run-time, and as an offline process, and allows for profile directed optimization to occur in the field (allowing the application to be customized to the end-user's usage pattern).

There are a number of weaknesses in the current approaches to compilation that we are trying to address. Some of the most prominant approaches to interprocedural optimization, profiling, and whole system optimization are:

- Perform interprocedural optimization at the source level. The primary drawback of this approach is the need for whole-program analysis phase at compile time, which disrupts the "compile-link-execute" build process (requiring significant changes to build scripts), and also substantially complicates compiler development.

- Perform interprocedural optimization at link time, using the compiler internal representation (IR) exported during static compilation [13, 3, 11], or using special annotations added to object files [29]. The main difficulties with this approach is that this extra information is proprietary and ad-hoc, making them difficult to use with third-party compilers. Additionally, runtime optimization using an exported compiler IR tends to be unfeasible because of the large size of most static intermediate representations.

- Perform interprocedural optimization on machine code in an "optimizing linker" [5, 12, 17, 23, 26, 10, 25]. This approach is transparent to the end user, but is limited by the lack of information provided by machine code. Machine code is a difficult medium for performing aggressive interprocedural optimizations, because the high-level structure of the code is obfuscated by low-level details of the machine. Because of this limitation, optimizing linkers typically are limited to low-level optimizations such as profile-driven code layout, interprocedural register allocation, constant propagation, and dead code elimination.

- Profile information is traditionally collected by creating a special “profile enabled” build of the application, running it with representative input, and feeding the data generated back into another build cycle. The primary problems with this approach are that it requires extra work for the developers to use (which reduces the likelyhood that profile directed compilation will be used), and that getting a representative input sample can be very difficult for some classes of application.

- Traditionally, once the application has been linked, the executable remains frozen throughout its lifetime. While speculative, we believe there may be use for *runtime* and *offline* optimization, which are not available in existing systems. Program transformation extremely late in the lifetime of the application is neccesary to adapt to changing usage patterns of the end-user and to adapt to hardware changes that may occur (allowing retuning for a new CPU, for example).

In contrast, LLVM enables novel interprocedural transformations at link time, runtime, and even in the field. It fits the standard “compile-link-execute” build model, working as a drop-in replacement for an existing compiler in build scripts, easing deployment of new optimizers. LLVM can gather profiling information transparently to the programmer, *without* requiring a special profile build and “representative” training executions.

This paper describes the high level approach to compilation used by LLVM (Section 2), contains detailed information about the virtual instruction set used throughout the compilation process (Section 3), describes some of the transformations implemented in the LLVM infrastructure (Section 4), and describes key components of our infrastructure (Section 5). An online reference manual describes the individual LLVM instructions and tools in more detail [19].

2 The LLVM Approach

The centerpiece of the system is the LLVM virtual instruction set. It combines a low-level representation with *high-level* information about operands, by using Static Single Assignment (SSA) form and high-level, language independent type information (more detail is provided in Section 3). The RISC like, low-level instructions of LLVM allow for many traditional optimizations to be directly applied to LLVM code, while the high-level type information allows for novel high-level transformations as well. This common representation enables the powerful multi-phase compilation strategy shown in Figure 1.

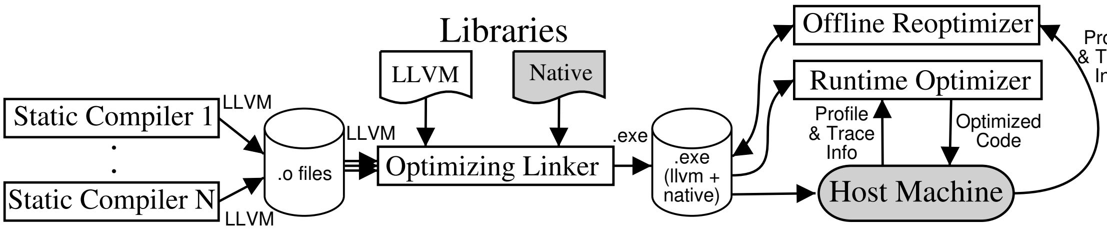

Figure 1: LLVM Compilation Strategy Overview

As Figure 1 illustrates, the LLVM compilation strategy exactly matches the standard compile-link-execute model of program development, with the addition of a runtime and offline optimizer. Unlike a traditional compiler, however, the `.o` files generated by an LLVM static compiler do not contain any machine code at all – they contain LLVM code in a compressed format.

The LLVM optimizing linker combines LLVM object files, applies interprocedural optimizations, generates native code, and links in libraries provided in native code form. An interesting feature of the executables produced by the optimizing linker is that they *contain* compressed LLVM bytecode in a special section of the executable itself. More information about the design and implementation of the optimizing linker is provided in Section 5.6.3.

Once the executable has been produced, the developers (or end-users) of the application begin executing the program. As they do so, trace and profile information generated by the execution can be optionally used by the runtime optimizer to transparently tune the generated executable. The runtime optimizer is decribed in Section 5.8.

As the runtime optimizer collects profile information and traces, it will determine that transformations need to be performed on the program. Unfortunately some transformations, especially interprocedural ones, cannot be profitably performed at runtime. For this reason, the offline reoptimizer is available to make use of spare cycles on the machine to perform agressive reoptimization of the application code *in the field*, tuning it to real world usage patterns. Our plans for the offline reoptimizer are discussed in Section 5.9.

This system heavily depends on having a code representation with the following qualities:

- The code must be high-level and expressive enough to permit high-level analyses and transformations at linktime and in the offline optimizer when profile information is available. Without the ability to perform high-level transformations, the representation does not provide any advantages over optimizing machine code directly.

- The code must have a dense representation, to avoid inflating native executables. Additionally, it is useful if the representation allows for random access to portions of the application code, allowing the runtime optimizer to avoid reading the entire application code into memory to do local transformations.

- The code must be low-level enough to perform lightweight transformations at runtime without too much overhead. If the code is low-level enough, code runtime code generation can be cheap enough to be feasible in a broad variety of situations. A low-level representation is also useful because it allows many traditional optimizations to be implemented without difficulty.

Section 3 describes the attributes of the LLVM virtual instruction set that make it suitable for this purpose.

3 The LLVM Virtual Instruction Set

The LLVM virtual instruction set is designed to be a *low-level* representation[1] with support for *high-level* analyses and transformations. To meet these goals, it provides extensive language independent type information about all values in the program, exposes memory allocation directly to the compiler, and is specifically designed to have few special cases. This section first provides an overview of the LLVM architecture, and then discusses the major features of the LLVM virtual instruction set. The detailed syntax and semantics of each instruction are defined in the online LLVM reference manual [19].

3.1 Overview of the LLVM architecture

The LLVM instruction set represents a virtual architecture that captures the key operations of ordinary processors, but avoids machine specific constraints such as physical registers, pipelines, low-level calling conventions, or traps. LLVM provides an infinite set of typed virtual registers, which can hold values of primitive types (integral, floating point, or pointer values). The virtual registers are in Static Single Assignment (SSA) form, which is a widely used representation for compiler optimization, as explained in Section 3.3. The LLVM type system is explained in more detail in Section 3.4.

[1]Note that Java bytecode and Microsoft's .NET bytecode both represent code at a much higher level than LLVM. In fact, the current LLVM system could be used to compile and optimize the virtual machines implementing these systems (which are often written in C and C++), rather than Java or .NET applications themselves. It is also possible to support compiling application bytecode to LLVM, which would allow using our optimization capabilities in the context of a Java static compiler, for example. For more details, see Section 5.6.

LLVM programs transfer values between virtual registers and memory solely via `load` and `store` operations, using typed pointers. Memory is partitioned into a global area, stack, and heap (where functions in the program are treated as global objects). Objects on the stack and heap are allocated using `alloca` and `malloc` instructions respectively, and are accessed through the pointer values returned by these operations. Stack objects are allocated in the stack frame of the current function, and are automatically freed when control leaves the function. Heap objects must be explicitly freed using a `free` instruction. The motivation and implementation of these operations are explained in Section 3.7 below.

Note that LLVM is a virtual instruction set: it does not define runtime and operating system functions such as I/O, memory management (in particular, garbage collection), signals, and many others. Some issues that do belong in the instruction set and may be added in the future include support for multimedia instructions (e.g., packed operations), predicated instructions, and explicit parallelism.

3.2 Three address code

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

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

A key point is that LLVM instructions are polymorphic, i.e., a single instruction like `add` can operate on several different types of operands. This greatly reduces the number of distinct opcodes. In particular, we do not require different opcodes for operations on signed and unsigned integers, single or double-precision floating point values, arithmetic or logical shifts, etc. The types of the operands automatically define the semantics of the operation and the type of the result, and must follow strict type rules defined in the reference manual [19].

For example, here are some simple LLVM operations:

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

3.3 Static Single Assignment form

LLVM uses Static Single Assignment (SSA) form as its primary code representation. A program is said to be in SSA form if each of its variables is defined exactly once, and each use of a variable is dominated by that variable's definition. SSA form greatly simplifies many dataflow optimizations because only a single definition can reach a particular use of a value, and finding that definition is trivial. It also enables fast flow-insensitive algorithms to achieve many of the benefits of flow-sensitive algorithms (sometimes referred to as the sparseness property).

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

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

SSA ϕ-node. The syntax of this instruction is:

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

`result` is assigned the value `val0` if control reaches this instruction from the basic block labelled `label0`, `val1` if control reaches here from basic block `label1`, and so on. All the `phi` instructions in a basic block must appear at the beginning of the basic block. For example the loop of this C function:

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

is translated to LLVM as:

```
Loop:
  %result  = phi  int [ %nextres, %Loop ], [ 1, %LoopHeader ]
  %i       = phi uint [ %nexti,   %Loop ], [ 0, %LoopHeader ]
  %nextres = mul int %result, %M              ; Result *= M
  %nexti   = add uint %i, 1                   ; i = i + 1
  %cond    = setne uint %nexti, %N            ; i != N
  br bool %cond2, label %Loop, label %Exit
```

As noted before, the virtual registers in LLVM are in SSA form while values in memory are not. This simplifies transformations (because scalars cannot have aliases), at the slight expense of complications for load and store motion (reordering load and store operations still requires pointer and array dependence analysis).

3.4 Type information

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

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

Some examples illustrating the types in LLVM are:

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

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


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

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

All LLVM instructions are strictly typed, and all have restrictions on their operands to simplify transformations and preserve type correctness (Additionally, these restrictions make the LLVM code more compact, as explained in Section 3.2). For example, the `add` instruction requires that both operands are of the same type, which must be an arithmetic (i.e., integral or floating-point) type, and it produces a value of that type. The `load` instruction requires a pointer operand to load from. The `store` instruction requires a value of some type (say, τ) to store and a pointer to store into, which must be a pointer to that type (τ*). Some examples of malformed code are:

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

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

3.5 Type-safe pointer arithmetic: the `getelementptr` instruction

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

The example below is a complex C testcase, designed to be a concise illustration of the LLVM lexical structure, type system, and the `getelementptr` instruction. The testcase defines two structure types and a function that performs complex indexing:

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

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

The LLVM code below is a version of the code generated by our C front-end, with LLVM comments added:

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

; Define function 'foo', returning an 'int*', taking an 'ST*':
int* %foo(%ST* %s) {
  ; Perform the indexing...
```

```
  %tmp = getelementptr %ST* %s, uint 1, ubyte 2, ubyte 1, uint 5, uint 13
  ret int* %tmp         ; Return the computed value
}
```

This LLVM code illustrates that LLVM identifiers (type and value names) start with a `%` character (to prevent namespace collisions with reserved words), shows some examples of complicated nested types, and shows an LLVM function definition. Additionally, it shows how useful named types are for hand inspection of code (without symbolic names provided by the C compiler, the types would all be expanded out inline, making them less managable).

3.6 Distinguishing safe and unsafe code: the `cast` instruction

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

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

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

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

Because LLVM is intended as a general-purpose low-level instruction set, it must represent both "type-safe" and "type-unsafe' programs for arbitrary high-level languages[2]. Nevertheless, distinguishing between safe and unsafe operations is important because many memory-oriented optimizations may only be legal for safe programs.

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

Some unsafe programs by this definition are truly unsafe: their results depend on the specific layout of their data structures in memory. Unfortunately, there is one common class of programs where this definition is too conservative: many languages implement container types (e.g., a hashtable in C) with generic reference types (e.g., `void*` in C), which need to be cast to the actual reference type as they are accessed. Such programs may be quite safe but requires more analysis to prove as safe. Without such an analysis, the cast operations cannot be eliminated and the program must be treated as unsafe. We have a future project planned to extend the LLVM type system to eliminate this deficiency.

If a program is type-safe by the above definition, type information can be exploited during important analyses such as alias analysis, and data structure reorganization transformations can be safely applied to it[3]. If a program is completely type-safe, its LLVM code can use the `getelementptr` instruction for all pointer arithmetic, without

[2]This is one of the ways LLVM differs from Typed Assembly Language[21], which focuses on program safety and therefore does not support type-unsafe programs, whereas LLVM is aimed at both optimization performance and program safety.

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

requiring any unsafe casts. For example, given a language with pointer arithmetic, naive compilation can cause type violations. For example, consider this C code:

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

The pointer arithmetic in the last statement could be compiled initially into this snippet of LLVM code:

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

The `%P2` cast is not type-safe because an arbitrary value is being cast to a pointer, and although in this case the result happens to be a valid integer pointer, the compiler cannot know that without further analysis. In the case above, however, simple induction variable analysis can convert the code to use array subscripting, or the `getelementptr` instruction could be used to directly navigate the array in a type-safe[4] manner:

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

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

3.7 Explicit Memory Allocation

Some of the hardest programs to adequately optimize are memory bound programs that make extensive use of the heap and complex data structures. To better expose memory allocation patterns to the compiler, we have added typed memory allocation instructions to the instruction set. The `malloc` instruction allocates one or more elements of a specific type on the heap, returning a typed pointer to the new memory. The `free` instruction releases the memory allocated through the `malloc` instruction[5]. The `alloca` instruction is similar to `malloc` except that it allocates memory in the stack frame of the current function instead of the heap, and the memory is automatically deallocated on return from the function.

These instructions are essential for preserving the type-safety of our representation (the normal C `malloc()` function returns an untyped pointer that must be cast to the appropriate type), and have enabled new transformations that would be very difficult without them (for example, *safe* automatic pool allocation and data structure reorganization for C programs [20]).

3.8 Function calls and exceptions

LLVM provides two function call instructions, which abstract away the calling conventions of the underlying machine, simplify program analysis, and provide support for exception handling. The simple `call` instruction takes a pointer to a function to call, as well as the arguments to pass in (which are passed by value). Although all `call` instructions

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

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

take a function pointer to invoke (and are thus seemingly indirect calls), for direct calls, the argument is a global constant (the address of a function). This common case can easily be identified simply by checking if the pointer is a global constant. The second function call instruction provided by LLVM is the `invoke` instruction, which is used for languages with exception handling.

LLVM implements a stack unwinding [24] 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 function pointer and argument operands of a normal `call` instruction. When code generation occurs, the return address of an invoke instruction is associated with the exception handler label specified, allowing the exception handling/cleanup routine to be invoked when the stack frame is unwound.

The `invoke` instruction is capable of representing high-level exceptions directly in LLVM, using only low-level concepts (return address to handler map). This also makes LLVM independent of the source 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 example illustrates the `invoke` instruction in the context of C++:

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

The key thing to note with this very simple example is that C++ guarantees that the destructors of stack allocated objects will be invoked if an exception is propogated through the current stack frame. Translated to LLVM, this becomes:

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

The `invoke` instruction associates an exception handler to call if an exception is propogated through the invoked function. In this example, this is used to invoke the destructor of a local object. In the context of the Java language (which does not have to call destructors when unwinding), the `invoke` instruction is used to unlock locks that are aquired through synchronized blocks or methods. In any language, a `catch` clause would be implemented in terms of an exception destination.

Although we currently do not have a front-end that uses the exception handling support built into LLVM, all of our optimizations and transformations are aware of the exceptional control flow edges, and our unit tests work as designed.

3.9 Few special cases

Being able to design our representation without concern for legacy compiler code has given us the ability to design a simple orthogonal instruction set with few special cases to consider. Special cases in representations complicate transformations in subtle ways that make them difficult to write, and subject to nefarious bugs. Some examples of special-case features that are unnecessary in LLVM are an "address-of" operator and an "indirect call" instruction.

In LLVM, all addressable objects (stack allocated locals, global variables, functions, dynamically allocated memory) are all explicitly allocated. Stack allocated locals are all explicitly allocated using the `alloca` instruction (which is the analog of `malloc` that allocates memory from the stack instead of the heap). Functions and global variables (collectively referred to as "global values") declare regions of statically allocated memory that are accessed though the address of the object (the name of the global value refers to the address). Heap allocated memory is obviously allocated with the `malloc` instruction. Because memory objects are always refered to by their address, an "address-of" operator is simply unneccesary. This representation also simplifies memory access analysis, since there cannot be implicit accesses to memory. All memory traffic occurs when "load" and "store" instructions execute.

As mentioned in Section 3.8, LLVM also does not have an "indirect call" instruction in addition to the normal `call` instruction. This is because the standard `call` instruction takes a pointer to function argument along with the rest of the arguments to pass to the function, instead of a function as a special part of the instruction. Effectively this means that all function calls are represented as indirect calls, although most of them have a function address explicitly passed in as the first argument.

We chose to make the default capabilities of the `call` instruction allow for indirect calls, to reduce the size of the instruction set (we would have to add an "indirect call" as well as an "indirect invoke" instruction to allow for generic indirect calls in the naive manner), and encourage transformations to implement the general case before the special case (which remains easy to support). This example shows how global values (in this case functions) are treated as pointers to their declared type:

```
void %func(int) { ... }

void %calltest(void (int)* %FuncPtr, int %Val) {
  ; Call the "func" function directly, passing in Val as its argument...
  call void (int)* %func(int %Val)

  ; Call the function pointer, passing in Val as its argument...
  call void (int)* %FuncPtr(int %Val)
  ret void
}

int %main() {
   ; Invoke the "calltest" function, passing in "func" as a function pointer...
   call void (void(int)*, int)* %calltest(void (int)* %func, int 42)
   ret int 0
}
```

Many other small features such as the two above combine together to make it significantly simpler to write and maintain analyses and transformations for a variety of purposes. Because of this, our compiler infrastructure is easily extended and has developed powerful features very rapidly.

4 LLVM Transformations

LLVM is a rapidly maturing infrastructure for compiler development and research. Many LLVM-to-LLVM transformations have been implemented so far (in very little time), showing that the LLVM infrastructure is powerful and easy to use. This section describes some of the more interesting transformations that are available as part of the LLVM infrastructure.

4.1 Traditional Data Flow Transformations

LLVM provides many traditional data flow transformations in addition to the more aggressive transformations described below. The SSA properties and simple three address form of LLVM make it very straightforward to implement simple optimizations like Dead Code Elimination, Constant Propagation, and Expression Reassociation. The SSA representation also enables classical SSA scalar transformations such as Sparse Conditional Constant Propagation and Aggressive Dead Code Elimination to run efficiently.

Through the use of the control flow graph (CFG) for a function, the LLVM infrastructure can calculate dominance information of various sorts, natural loop nesting information, and even interval information for a transformation as desired. This information makes code motion transformations such as Global Common Subexpression Elimination, Loop Invariant Code Motion, and induction variable transformations simple to implement by building on a solid analysis foundation. LLVM also provides a pass that simplifies the CFG for the function, removing trivially dead blocks, merging blocks if possible, and performing other cleanup activities.

There are several reasons that we have reimplemented these common and well known transformations in LLVM. First, writing transformations is instructive and tests the generality of the infrastructure being developed. Second, these transformations are useful for cleaning up the result of running some of our aggressive transformations which concentrate on large scalar restructuring, often leaving little details to be cleaned up by these passes.

Finally, we have found that our implementation of these passes is much more efficient and effective than the similar passes in the GCC compiler we have retargeted to LLVM (See Section 5.6 for more information). By using sparse optimizations on SSA form, these transformations are much more efficient, while also having global effects. Several of the transformations in GCC operate on extended basic blocks, limiting their scope for improvement.

4.2 The Level Raising Pass

Many existing compilers do not retain type information for the program once they convert from a high-level code representation (e.g., an Abstract Syntax Tree or AST) to a lower-level representation used for optimizations. Because retargeting existing compilers is an important part of the overall LLVM strategy, we developed the "level raising" pass to assist with reconstructing this type information. With the level raising pass, compiler front-ends can be modified to simply generate *legal* LLVM code that is simple but low-level and not type-safe (e.g., using explicit byte addressing for structure field accesses). The backend can use the level raising pass to recover type information (e.g., direct references to structure fields) through program analysis. In this way, the program analysis required to recover type information can be shared by many preexisting compiler frontends without any integration problems: it is simply another LLVM to LLVM transformation.

The "level raising" pass operates by eliminating `cast` instructions from the LLVM code through several different strategies. It assumes that the prototypes for the functions are correct, and proceeds by eliminating `cast` instructions until its analysis cannot prove the remaining `cast` instructions away. In our current front-end, the prototypes for functions are easily accessable from the debug information maintained by the compiler. On the other hand, the internal representation of the function itself may be arbitrarily modified by optimizations in the compiler, making instruction level type information virtually impossible to reconstruct. Here is an example of the transformations made by the level raising pass:

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

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

From this C snippet, our GCC front-end outputs the following correct, but suboptimal LLVM code (the strength reduction of the multiply was done by GCC optimizations, not by LLVM):

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

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

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

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

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

...which is now type-safe.

4.3 Simple Memory Transformations

One of the focuses of LLVM from the beginning has been to enable and simplify memory access transformations, because memory is one of the main bottlenecks for both architectures and compiler analysis. Two examples of simple memory optimizations that have been implemented in LLVM are the `mem2reg` and various structure permutation passes.

The `mem2reg` pass promotes stack allocated memory objects to live in SSA registers. The implementation first analyzes the stack allocated objects in a function to see if they are safe to promote (i.e. their address is not required to exist), then performs a slightly simplified SSA construction pass to place ϕ-nodes in the function. Once in SSA form, `load` and `store` instructions are rewritten to be register copies. This transformation illustrates the importance of type information in our representation. Without type information, this transformation would be impossible to perform, because in general, stack allocated objects can be arbitrarily complex (complex nested structures, for example).

LLVM also contains various structure permutation transformations which can be used to improve the cache behavior of the program. These transformations first analyze the usage of data structures to ensure they are accessed in a type-safe manner. If it is safe to do so, structure definitions are modified, and the program is transformed to load and store from the fields of the transformed structure definitions. Using the type information available to the LLVM compiler, and the type-safety information made explicit through the `cast` instruction, these aggressive transformations on C programs are straightforward to implement.

4.4 Aggressive Data Structure Transformations

A recent research result of the LLVM project has been a framework for aggressive data structure transformations [20]. Using information about memory allocation and typed memory operations, we describe how to build a "data structure analysis graph". This graph describes the logical, *disjoint*, data structures (hash table, AST, binary tree, graph, etc...) that are used by the program, and whether or not those data structures are accessed in a type-safe manner.

Given this graph, it is possible to perform a *fully automatic* pool allocation transformation on the program, which converts the program from using general heap allocation to use a carefully controlled pool allocation library. This transformation is nontrivial because the source language is C, the data structures are heavily recursive, and we do a precise, data structure based pool allocation. Because the transformation is precise, indicating that exact analysis is used instead of heuristics to allocate nodes to pools, the pool allocation can be used as the basis for other, more aggressive data structure transformations.

Applications of automatic pool allocation are varied: everything from data structure based dead-field-elimination, to virtually free allocation order prefetching, to a technique we refer to as *pointer compression*. Although applications of automatic pool allocation are still being investigated, the ability to perform *macroscopic* data-structure-centric transformations on a program at *link-time* is a single example of the possibilities that the LLVM system enables.

4.5 Instrumentation

LLVM provides two main "instrumentation" passes. The first is primarily a debugging aid for the native code generator (`llc`). This pass inserts code into the LLVM program that causes it to print out the intermediate values produced by individual instructions or individual function calls in the program. `llc` invokes this pass to instrument the "final" LLVM code, just before beginning native code generation, and outputs a copy of the instrumented LLVM code. The output of the instrumented program when run in `lli` should match the output of the native code generated by `llc` exactly. This instrumentation pass features a deterministic pointer hashing scheme that allows it to print out identical values for pointer variables in `lli` and any machine on which the code might be executed. If the native code is incorrect, the two outputs can be used to identify the first few instructions that produced a different value in the LLVM code and the native code. This can greatly simplify debugging the generated native code.

The other important instrumentation pass is used to add code to collect "path profiling" [6] information from the program. Path profiling is intended to be used by the runtime and offline optimizers (see Sections 5.8 and 5.9 respectively) to identify hot traces of the program execution. Although more expensive to compute than edge profiling information, path profiling provides important information about the frequency of execution of traces of the program, rather than simple execution counts for basic blocks. Using LLVM as the target (instead of machine code) for the instrumentation simplifies the implementation tremendously (there is no need to worry about the availablity of registers, for example), and also allows the profiling code to be optimized with the rest of the program in LLVM form.

5 The LLVM Tool Chain

In addition to providing a suite of tranformations, the LLVM infrastructure also provides a number of executable "tools" that are useful for interacting with the LLVM system. One of the key design goals of the LLVM project is to make LLVM completely transparent to the end user (a programmer compiling with LLVM). As such, most of these tools are for developing or debugging LLVM transformations, and are therefore LLVM developer tools. This section describes the design of the main components of our infrastructure and their current implementation status.

5.1 The LLVM Assembler and Disassembler

One critical, but often overlooked, aspect of a code representation is the form that the representation takes. The LLVM virtual instruction set is itself a first class language, complete with a textual format (examples of which have been included in this document) that is equivalent to the in memory and "bytecode" (high-density) representation. This allows for straightforward debugging, as well as a simple unit test framework (because test-cases can be written directly in LLVM instead of C). Although the code stays in the bytecode representation most of the time, it is easy to use the disassembler to convert the code to LLVM assembly and back, losslessly.

In order to support the textual form of LLVM, we have simple assembler (`as`) and disassembler (`dis`) tools that convert to and from the compressed LLVM bytecode representation and textual forms. The disassembler is also capable of converting the LLVM code to C, which is useful for people familiarizing themselves with the LLVM system, and for targeting systems we do not have a code generator for.

5.2 LLVM Interpreter/Debugger/Profiler: `lli`

The LLVM interpreter, `lli`, is a general purpose LLVM execution tool. It is capable of directly interpreting compressed LLVM code without a code generation pass (and emulates several common system calls to support I/O). The LLVM interpreter has a built in debugging mode, allowing for instruction-by-instruction stepping through the LLVM instruction stream, stack-traces, and other features expected from a debugger. The profiler mode of `lli` can collect statistics about the dynamic number of instructions executed by a program, and the dynamic count of certain program events.

`lli` is very slow (approximately 1000 times slower than native execution), because it is designed for flexibility and ease of implementation, not performance. In practice, we use `lli` to pinpoint the location of problems in the LLVM system: If the code executes properly in `lli`, but not after native code-generation, then the bug must be in the back-end, otherwise it's in an optimization transformation. Because `lli` is used infrequently, its performance has not been a problem in practice.

5.3 LLVM Modular Optimizer: `opt`

The modular optimizer, `opt`, is the centerpiece of development and developer interaction in the LLVM infrastructure. It exposes all optimizations to the user through a simple command line interface which can execute arbitrary sequences of transformations on a program. This is critical for debugging a transformation, so that the programmer can find out what changes her pass is making without having to worry about what all of the other transformations in the compiler are doing. In addition to simply running transformations, the modular optimizer uses the LLVM verifier to check that output code from a transformation is still well formed (the verifier can, for example, detect type violations, malformed SSA code, etc). Once a transformation has been completed and debugged, it can be easily integrated into the compiler proper.

The `opt` tool exposes many transformations available in LLVM. Many of them (such as `DCE`, `constprop`, `gcse`, `licm`, `inline`, `simplifycfg`, `indvars`, `sccp`, `adce`, and `reassociate`) are traditional mid-level optimizations. In addition to traditional optimizations, the LLVM system also provides a variety of other interesting transformations, some of which are described below.

5.4 Testing framework

Given a modular optimizer and many transformations, a natural extension is to develop a suite of test cases for the transformations. We have developed a suite of unit tests (to determine whether an optimization does what it is supposed to), regression tests (to ensure bugs stay fixed), and whole system tests. These whole system tests are designed to take various programs (benchmarks, programs taken from the Internet, and other interesting programs), compile them with the LLVM compiler (for direct use with the LLVM interpreter), to native code with one of the LLVM back-ends, and to native code with the platform native compiler, executing the results and '`diff`'ing the output. This testing framework helps us track down regressions quickly and effectively, allowing us to test new transformations thoroughly before they are added to the compiler proper, and ensuring that the mainline compiler works well on a variety of codes.

5.5 The LLVM Optimizing Linker

The structure of the LLVM optimizing linker is shown in Figure 2 (When invoked from GCC, this linker is invoked via the name `gccld` as explained in Section 5.6.3, but is entirely independent of GCC). The linker links together LLVM object files, but can also perform aggressive interprocedural post-link optimizations. The link-time optimizer is fully operational and has already been the host of some interesting research, including the data structure analysis and the automatic pool allocation and pointer compression transformations described in Section 4.4.

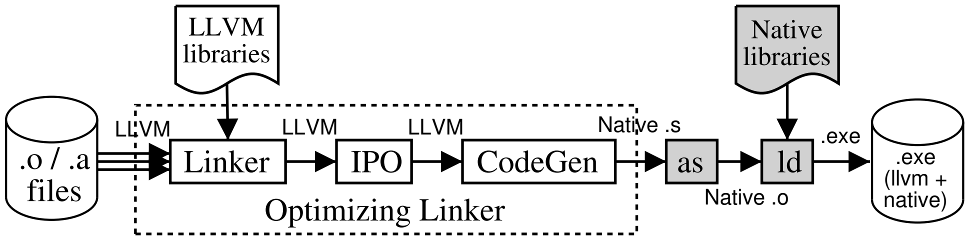

Figure 2: LLVM Optimizing Linker

It is important to note that many standard intraprocedural optimizations need only be performed on individual `.o` files when the corresponding source files are modified. Such optimizations can be invoked either by the final phase of the retargeted front-end compiler (e.g., `gccas`) or by the optimizing linker *before* the linking pass.

Because the optimizing linker can perform transformations at post-link time, when the entire program is available for analysis and transformation, it makes interprocedural transformations relatively simple to implement. Because the LLVM transformations are modular, we can use the traditional scalar optimizations described in Section 4.1 to clean up any inefficiencies introduced by the interprocedural transformations, providing all of the normal benefits of the "separation of concerns" approach to compiler design [2]. The current LLVM infrastructure shown in Figure 2 reflects this strategy.

A more sophisticated interprocedural strategy is to perform the initial intraprocedural information-gathering phases on a per-translation-unit basis (similar to the intraprocedural optimizations described above), and use these as the starting point for interprocedural optimizations during the IPO phase. A persistent program database would be needed to record the results of the initial intraprocedural analyses. This ensures that the initial per-module analyses would only have to be performed on modules that have been recompiled, but it adds some complexity to the compiler infrastructure. (Note, however, that this approach would still be completely compatible with existing build environments, unlike source-level interprocedural systems). The current LLVM infrastructure only implements the simpler strategy, but the more sophisticated strategy may be added in the future.

5.6 GCC based C Static Compiler

The first pre-existing compiler that we have retargeted to generate LLVM code is the GCC 3.0 (the Gnu Compiler Collection) C compiler[6]. This gives us excellent support for many pre-ANSI as well as ANSI-compatible benchmarks, the broad range of GNU extensions, as well as compatibility with pre-existing makefiles. In particular, we automatically support the GCC command-line options, thus making the existence of LLVM transparent to an ordinary user of GCC. However, we do have one limitation: we do not currently generate symbol-table information for use by debuggers such as GDB. The organization of the LLVM GCC front-end is shown in Figure 3.

As with many compilers, the `gcc` command is actually a driver program that does not do any compilation. Depending on the command-line options provided, it may `fork` and `exec` various subcomponents of the compiler,

[6]We are planning to add support for C++ is soon, and Java support will be added as time allows. The most complex part of enabling Java support is working out the runtime issues.

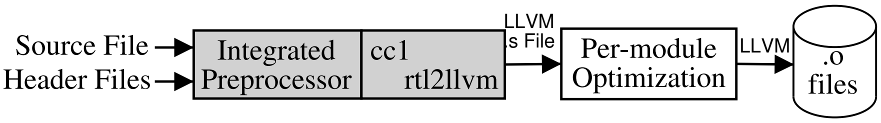

Figure 3: GCC C Front-end Compiler Organization

which include the preprocessor (which is actually integrated in with the C front-end in this release of GCC), the C front-end, the assembler, and the linker.

Our port required modifying the GCC compiler back-end (which is part of the `cc1` executable) to emit LLVM code (described in Section 5.6.1), implementing an assembler for GCC to invoke (`gccas`, described in Section 5.6.2), and implementing a linker for GCC to invoke (`gccld`, described in Section 5.6.3). As expected, the preprocessor and C parser were not modified.

5.6.1 Modifications to the GCC back-end: `cc1`

Normally, GCC uses two intermediate representations for its internal compilation processes, the "tree" representation (an Abstract Syntax Tree), and the Register Transfer Language (a low-level, untyped, machine independent representation). GCC compiles a function at a time into the tree representation, performs optimization on it (including source language specific optimizations), then uses a target-dependent machine description file to transform tree-code into target-specific RTL code. Once in RTL form, traditional scalar optimizations are used to optimize the program, including an experimental SSA based dead code elimination pass.

For our port, we wanted to be as minimally invasive in the GCC source-base as possible, while being able to reuse most of the GCC infrastructure. As such, we decided that it would be most appropriate to convert RTL to LLVM code instead of the tree representation, allowing us to make use of the scalar optimizations already in GCC. We wrote a simple machine description file, describing a simple orthogonal machine free of peculiarities that would unduly complicate the generated RTL. Because LLVM is unlike normal targets (it is SSA based, and doesn't require register allocation), we changed the compiler to emit LLVM code after SSA based dead-code elimination.

One disadvantage of converting RTL to LLVM (as opposed to the tree representation), is the RTL is untyped. Because of this, we only know the size of integral data-types, not their sign. Additionally, we have no information about what type a pointer refers to, and all pointer arithmetic has been lowered to explicit arithmetic. We do, however, have access to type information for function arguments, global variables, and typedef information (by adding "debugging" hooks to GCC), and some information that was implicitly encoded into the instructions of the RTL code (for example, information about sign is encoded into the choice of a signed or unsigned divide instruction).

Due to these limitations, almost all pointers in the program compiled by the LLVM GCC front-end are plain byte pointers, completely lacking high level information. We suspect that other retargeted compilers would have similar problems, because most optimization happens on low-level representations. To compensate for this lack of information, the GCC front-end encodes as much type information as possible by inserting casts where it knows that something is signed or unsigned, and by emitting accurate function prototypes and global variable declarations. It depends on the optimizing assembler to clean up the rest of the type information.

5.6.2 The Optimizing Assembler: `gccas`

The assembler is invoked by GCC to turn a textual LLVM program into an LLVM object file (which is the compressed bytecode representation). This involves more tasks than the standard LLVM `as` utility mentioned in Section 5.1, for several reasons. The two reasons for `gccas` to be distinct from `as` was that the GCC driver expected the assembler to provide command line arguments different than the arguments expected by `as`, and the fact that we wanted to invoke LLVM infrastructure passes on the code generated by GCC (for example, the "level raising" transformation described in Section 4.2).

We also found that many of the scalar optimizations implemented directly in LLVM are faster or are more effective than those built into GCC (because we perform all of our optimizations in a sparse manner on SSA form instead of using dense data-flow analysis, and because they are all global transformations, whereas some of GCC's operate on extended basic blocks). Thus we have been able to reduce compilation time by disabling them in GCC and enabling the equivalent transformation in `gccas`. In addition, several LLVM passes have no analog in the GCC compiler (such as our generalized `mem2reg` pass, described in Section 4.3), but we would like to use then for their general code improvement capabilities.

Note that when porting a different compiler with more aggressive optimizations to LLVM, it would be important to revisit the decision of which optimizations to move to the equivalent of `gccas` for that compiler. Because `gccas` is implemented by simply invoking transformations from a common library of transformations available in the LLVM infrastructure, it should be a simple matter of selecting the transformations to run and writing the command line option processing code to match the desired interface.

5.6.3 The Optimizing Linker: `gccld`

Like `gccas`, `gccld` is invoked by the GCC compiler driver, in this case, to link `.o` files together into a library or application. To ease integration with GCC, `gccld` provides a set of command-line options compatible with traditional linkers (such as the `-llibrary`, `-Lpath`, and `-s` options). It links together the LLVM object files, but can also perform aggressive post-link optimizations as shown in Figure 2. The optimizing linker was described in detail in Section 5.5.

5.7 Native Code Generator for Sparc v9

Our first native code generator targets the 64-bit Sparc v9 architecture. It includes an instruction selector based on BURG [15], a graph-coloring register allocator, and a resource-unit-based local instruction scheduler. All three phases have been designed to relatively easy to retarget, as discussed at the end of this subsection below.

BURG operates on a tree representation of a procedure and uses machine-dependent patterns and associated costs to select optimal instruction sequences via dynamic programming. The instruction selection phase in LLVM uses the SSA graph to construct BURG trees. A BURG tree is a tree of instructions, where the children of an instruction, *I*, are those that compute the operands of *I*, and the parent (if any) is some instruction that uses the result of *I*. In the SSA graph and in real code, however, an instruction may have multiple users, and at most one user can be represented in the tree (This need for trees in pattern-matching rather than DAGs is one of the major limitations of the BURG approach).

We convert the graph to a tree based on the following observation: The fundamental optimization that exploits tree edges is to produce the lowest cost sequence for two (or more) instructions that produce and then consume a value or a sequence of values. The most common example is folding two such instructions into a single machine instruction. If the result of instruction *I* is used by multiple instructions, it cannot be folded with any of them. Therefore, we do not connect such an instruction to its users. It is also difficult to fold two instructions from different basic blocks. Therefore, we build BURG trees and perform instruction selection separately for each basic block. The BURG patterns and costs are based on cost estimates for the Ultra-SPARC IIi processor [27].

The local instruction scheduling phase uses a forward list scheduling algorithm on individual basic blocks [22]. The algorithm assumes a statically scheduled super-scalar architecture and uses a greedy grouping of instructions in the slots for each cycle. The algorithm builds a dependence graph for the machine instructions generated by the selection phase, using SSA information to greatly speed up the detection of most dependences (since register allocation has not yet been performed, instruction operands are still in near-SSA form). Both the scheduling heuristics and the grouping are based on an extensive parametric description of the instruction costs, pipeline resources, and issue constraints of the target processor. This allows the compiler code itself to be entirely target-independent. The algorithm considers detailed issue constraints including restrictions on issue slots, grouping of instructions, pipeline bubbles produced by individual opcodes, and minimum inter-instruction issue delays (i.e., when a certain number of cycles must elapse between issuing one opcode and another).

Finally, the register allocation phase is a straightforward implementation of Chaitin's algorithm for graph coloring register allocation [9] as modified by Briggs et al (particularly for deferred spilling) [7]. We use a separate iterative data-flow analysis to compute the liveness for virtual registers. Once this is available, the SSA representation in LLVM makes it straightforward to build the interference graph. We use a set of parameters to describe register classes of the target machine, to keep most of the allocation algorithm machine-independent (Some machine-specific code is necessary as described below). We use assignment hints for specific values such as procedure arguments and return values to guide register allocation and assignment and reduce the need for copies.

Two key missing pieces in the current code generator are a peephole optimizer and a global instruction scheduler. We are currently developing a global scheduler based on the Swing algorithm for modulo scheduling [18]. Because the global scheduler must move code across basic blocks, a key design goal is to perform global scheduling on the LLVM code, using cost information for the target machine [7]. This is important for a novel runtime optimization strategy we are exploring, as discussed briefly in Section 5.8. A target-dependent peephole optimizer will be added in the near future.

We believe the code generator should be relatively easy to re-target to other similar architectures. The use of BURG for instruction selection makes this phase conceptually simple to retarget, but this phase still has the most machine-dependent code. Porting this phase to a new architecture requires modifying the BURG pattern rules and costs, and rewriting the machine specific code that synthesizes specific opcodes from LLVM instructions. The detailed parameterized resource model used for scheduling should be simple to re-target for other similar architectures. The register allocator uses a collection of target-specific instances of generic classes (which have a target-independent interface) to manage machine specific details such as providing register assignment hints and to generate code sequences for copying and spilling.

5.8 Runtime Re-optimizer

One of the key research goals of the LLVM project is to develop a new strategy for runtime optimization and novel optimization algorithms, by exploiting the LLVM code representation. Unlike other virtual machine based systems such as those for SmallTalk, Self, Java, or Microsoft CLR, the goal in LLVM is to generate native code-generation at link-time, and to perform runtime optimization on this machine code, using the LLVM representation as a source of analysis information.

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 procedures. Trace-based optimization focuses optimizations on frequently executed dynamic sequences of instructions called traces [14, 4]. The traces being gathered in our system only include one or more complete basic blocks, i.e., a trace is a dynamic sequence of basic blocks. A key property of our code generator is that every basic block of LLVM code maps to a single basic block of machine code. This means that a trace of machine code can be mapped directly to the corresponding trace in the LLVM code, which can then serve as a source of high-level information for optimization.

Procedure-based optimization is widely used in many adaptive JIT optimizers [16, 8, 28]. This focuses optimization effort on entire procedures that are frequently executed (or that consume a large fraction of execution time [1]). Procedure-based optimization can be used to perform more aggressive procedure-wide optimizations on the LLVM code based on runtime information.

The trace-based runtime optimizer is currently under development. We have implemented an instrumentation phase that inserts low overhead tracing code into LLVM code (described in Section 4.5), allowing extraction of hot traces from the program. The LLVM bytecode provided by the Sparc back-end is accessible to the runtime optimizer when reoptimization is needed. We are currently developing other key components including a trace cache, mechanisms to switch between trace-based and normal execution, and a runtime interface to the mapping information gathered by the

[7]Note that *none* of the code generation phases moves machine code across basic blocks.

native code generator. This system should then provide a novel platform for exploring new dynamic optimizations that are much more aggressive than those previously attempted in binary runtime optimizers.

5.9 Offline Re-optimizer

The trace generation infrastructure used in the runtime re-optimizer can also be augmented to form the basis for the offline re-optimizer. In addition to traces, we would like to use the path-profiling information for aggressive profile-directed optimizations, like those described in the post-link optimizer. Developing an offline reoptimization framework is a topic for future work.

6 Conclusion

This document has introduced the LLVM Compilation Infrastructure, a clean new approach for tackling today and tomorrow's difficult problems in compilation. It is designed as a seamlessly replacement for preexisting compilers, without requiring a change in end-user build scripts or user habits. Within the constraint of compatibility with pre-existing systems, we have shown how interesting transformations can be applied to real world programs and shown how runtime and offline optimization fit into the system as well. LLVM has already been used as the foundation for interesting research [20] and we look forward to future development.

References

[1] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, 2001.

[2] M. A. Auslander and M. E. Hopkins. An overview of the pl.8 compiler. In *Proc. ACM SIGPLAN 82 Symposium on Compiler Construction*, pages 22–31, June 1982.

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

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

[5] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2000.

[6] Thomas Ball and James R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.

[7] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[8] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, Vugranam C. Sreedhar, Harini Srinivasan, and John Whaley. The jalapeno dynamic optimizing compiler for java. In *Java Grande*, pages 129–141, 1999.

[9] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proc. SIGPLAN '82 Symposium on Compiler Construction*, June 1982.

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

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

[12] Kemal Ebcioglu and Erik R. Altman. DAISY: Dynamic compilation for 100In *ISCA*, pages 26–37, 1997.

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

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

[15] C.W. Fraser, R.R. Henry, and T.A. Proebsting. BURG - fast optimal instruction selection and tree parsing. *ACM Sigplan Notices*, 27(4):68–76, April 1992.

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

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

[18] E. Ayguadé J. Llosa, A. González and M. Valero. Swing modulo scheduling: A lifetime-sensitive approach. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, pages 80–86, October 1996.

[19] Chris Lattner. LLVM Assembly Language Reference Manual. `http://llvm.cs.uiuc.edu/docs/LangRef.html`, 2002.

[20] Chris Lattner and Vikram Adve. Analysis and optimizations for disjoint data structures. In *ACM SIGPLAN Workshop on Memory System Performance (MSP'02)*, June 2002.

[21] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system f to typed assembly language. *25th ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 85–97, January 1998.

[22] Steven P. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205., 1st edition, 1997.

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

[24] Norman Ramsey and Simon Peyton Jones. A single intermediate language that supports multiple implementations of exceptions. In *Proc. SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

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

[27] Sun Microsystems. UltraSPARC^{TM} IIi Users Manual, 1999.

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

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