LLVM简介

LLVM是一款对应用程序开发者透明的、终身程序分析的、可转换任意程序的编译器框架（Lifelong Program Analysis & Transformation）。它通过以下两点来达到上述要求：

1. 一个拥有一些新奇特性的“代码表示”来作为分析、翻译、代码分发的基础。

2. 一个利用上述“代码表示”提供的一组在以前编译方案中不从出现过的功能的来实现的编译器。

LLVM是一款很好适应现代编程语言和体系结构的编译器基础框架。LLVM所要到达的目标有如下重要三点：

1. 使用一个主动的（aggressive）、多阶段的优化策略，来使得程序性能最优。

2. 成为一个前沿研究开发的基点，为当前和将来研究提供一个坚实的基础。

3. 和系统编译器保持一致，使得操作对终端用户（程序开发者）透明。

与之前传统编译器区别

编译器开发者从各个方面出发，提出各种各样的方法来对程序进行优化，最终达到使程序性能最优的目的。一般来说，编译器开发者将利用如下3种技术中的一种或多种来开发编译器。然而，这些技术并不适合所有的编译器，并使之达到性能最优、编译时间最短的效果。

1. 连接时的过程间优化。

2. 运行时的动态优化。

3. 概要信息导向（profile-driven）优化。

对于以上3种技术，传统的编译器也采取了不同的策略，为了更好的理解LLVM，我们不妨简单的对传统编译器在使用这3种技术做一个了解：

（1）连接时的过程间优化

连接时的过程间优化是编译优化的关键一环，因为连接过程中将所有分别编译好的文件整合到一起，此时的优化是对程序的全局优化，也是优化效率最高的优化阶段。换句话说，连接时的过程间优化往往决定了一个编译器的优化效果的关键。而往往决定连接时优化的关键在于编译器是在哪一层做优化。是非常低层次的机器码层，还是非常高层次的抽象语法树层？

有一大部分研究者选择在连接时或者运行时在机器码上做过程间的优化。这样的一个好处在于它们在不被改变的前端编译器上工作的很好，可以是开发者使用任何他们想使用的编译器。但是这种编译器系统有很多的限制：机器代码不能够提供足够的高层信息来支持主动的过程中间分析和转换。

为了解决这个问题，一些开发者提出新的技术来保存源代码级的信息直到连接时。通常，是通过将高层编译器的中间表示在编译时写到硬盘上来实现的。在连接时，连接器通过读取一系列版本的抽象语法树，对其进行组合、优化，最终生成可执行代码。但是这种技术的代价是高昂的，因为几乎所有的编译工作都被延迟到连接时，这就导致任何的程序改变都将导致所有的程序都必须重新编译。

总上所述，传统编译器的连接时时优化，要么存在无法提供高层信息，要么代价高昂的问题，这些时现代编译器不能承受的。

（2）运行时的动态优化

对于运行时的动态优化，一个最为通用，也是最简单的方法就是直接完全忽略动态优化。然而，这毕竟是一种懒政,随着用户对程序性能的要求提升，这已不再是一个可以直接忽略的问题了。目前，运行时优化已经成为现代系统中一个频繁被使用的提高程序性能的技术。

提及运行时的动态优化，不能不提虚拟机。这里的虚拟机指的是高级语言运行时虚拟机（如：JVM等）。通过使用虚拟机和高层次的程序表示输入，这些系统能够提高可移植的、安全的服务来提高程序的性能，并且，它可以提供更多的、有用的高层信息给运行时优化器使用。不幸的是，高层次的连接时优化一样，这种系统使得编译时不能够做实质性的优化工作。例如，JIT编译器，这种动态编译器必须占用非常多的处理器周期来进行优化操作。

另一方面，机器代码级的运行时优化器以及指令翻译器通过直接操作本地机器代码来提高程序性能或者动态的翻译机器代码。很明显，这种方式有着和机器代码级过程间优化类似的弊端，在需要高精度的概要信息（profiling information）的轨迹生成和优化方面共工作的很好，但是对于高层次的重构转换应对不足。

（3）概要信息导向（Profile-driven）优化

概要信息导向优化是一种利用收集来的程序运行时行为信息来提高程序性能的一种技术。传统的概要信息导向优化需要经历五个阶段：

1. 编译程序，在此阶段，需要在程序中插入相应的指令，这些指令在运行时会被触发，以此来收集各种格式的概要信息。

2. 链接编译阶段生成的对象文件，使之成为可执行文件。

3. 开发者多次执行已生成的可执行文件，以此来收集概要信息

4. 和阶段5一起，重新编译和连接源程序，并且利用已收集的概要信息进行优化。

概要信息导向优化一种对程序信息有很大作用的技术，但是也存在着不少问题，如：

1. 概要信息必须精确才能够起作用。

2. 开发者在多次运行可执行程序来获取概要信息的运行环境和模式可能和最终应用程序使用者不相同。由于，这种传统的概要信息导向优化是一种静态优化，可能由于开发者采集的概要信息与实际使用者的实际信息不同而带来副作用。例如，可能使得程序在实际运行时更慢。

3. 更重要的是，由于这种优化相当的繁琐，开发者为了获取概要信息还的切换不同的编译和运行环境，而且还不一定得到预期的效果，更甚至相反的效果，这最终导致开发者不愿意做概要信息导向优化。

LLVM的5种功能

LLVM利用其代码表示提供了一组5种功能来实现程序终身分析和对任意程序的转换。而且这5种功能是其他编译器不能同时具备的。

1. 一致的程序信息（persistent program information）：LLVM编译器在软件整个生存期一直保存着LLVM汇编 代码，这使得可以在任何时期对代码进行分析和优化。

2. 编译型代码生成(offline code generation)：对于性能关键的程序，可以把程序直接编译为本地机器代 码。通过使用复杂的代码生成技术，可以保证生成的本地机器代码的高效 性。

3. 基于用户的概要信息和优化(user-based profiling and optimization)：LLVM在运行时向用户收集剖面信 息，并把这些信息应用到基于剖面的转换中。这样，软件就能反映用户的 具体使用习惯。

4. 透明的运行时模型(transparent runtime model)：LLVM不指定对象模型、异常语义或者运行时环境， 这使得LLVM能编译支持任何语言的程序的分析和转换。

5. 统一的全程序优化(uniform, whole-program compilation)：语言独立性使得LLVM可以以统一的方式编译和优化 程序的所有代码，包括系统库和语言特定的运行时库。

LLVM与高层虚拟机的区别

根据LLVM（Low Level Virtual Machine）的名字很容易让人觉得它是一个虚拟机，但是它事实上是一个模块化的、可重用的编译器和工具集合。它和传统的虚拟机关系不大，但是可以通过它提供的库来实现一个虚拟机。总之，LLVM不再是首字母缩写，它这个项目的全称。

由于设计目标和中间表示不同，LLVM （Low Level Virtual Machine）是高层次虚拟机（如：JVM、Microsoft CLI）的补充，而非另外的一种选择。它们的区别主要在于以下3点：

1. LLVM不考虑高级语法特性。如数据结构、类、继承、异常处理等。

2. LVMM不指定运行时系统或者特殊的对象模型。LLVM足够底层，它完全能自己实现指定语言的运行时系统。

3. LLVM不保证类型和访存安全以及语言间操作强于汇编语言。

LLVM系统架构

LLVM系统架构这是为了解决这些传统编译器所存在的问题而设计的。简单的说，在LLVM系统中静态编译器将源代码编译成低层次的表示（为方便后面将称为“中间表示”或IR或”LLVM代码表示“）——LLVM虚拟指令集并且包含了高层次的类型信息。这样静态编译将可以在编译时做实质性的优化并且可以保留高层信息提供给连接器使用。

LLVM是一个多阶段优化的编译器框架。这种策略的独到之处在于其可以在程序的整个生命周期内做优化。和传统编译器相比，LLVM可以在连接时、运行时甚至在程序安装之后做复杂的转换和分析。图-1 是LLVM的概要图。

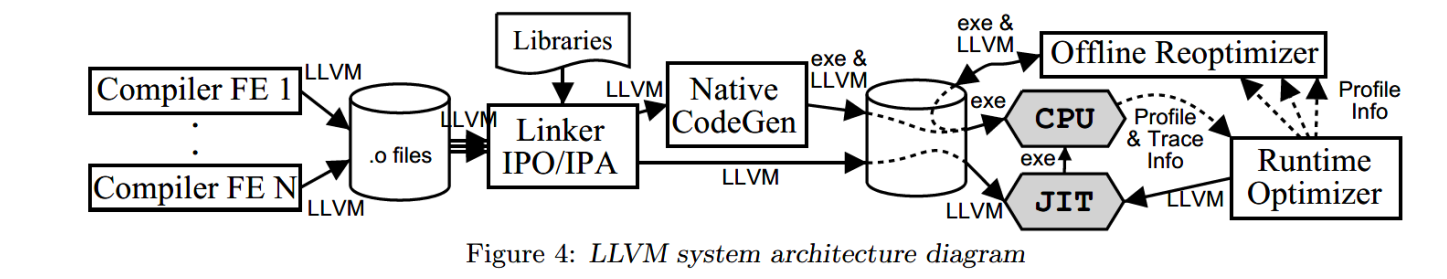


图-1 LLVM概要图

传统编译器只有两个阶段——编译和连接。LLVM依旧保留着这两个阶段。这样做的目的是为了利用分段编译的优点——当程序发生改变时只需重新编译被改变的部分。与传统编译器不同的是，传统编译器在编译阶段生成的.o文件已经是机器代码了，而LLVM则是将源代码翻译成LLVM虚拟指令集（就是IR，这些代码将和最终生成的本地机器代码一起保存，在运行时可以通过JIT翻译器对其进行翻译），而后由连接器将其连接并做优化，最终生成可执行的本地代码存到磁盘上。本地代码生成器可以将轻量级的指令插到本地机器代码中以此来频繁的探测循环、执行路径、函数调用，在运行时，可以以此来收集概要信息进行再优化。根据程序的行为，在运行时可以动态的再编译和再优化，然而这种优化有时候是代价巨大的，所有这部分优化可以在程序运行的空闲时间来完成。这便是LLVM的大概流程。

编译时：前端和静态优化器

LLVM编译器是支持多种编程语言前端的，每一种前端都必须将源程序翻译成LLVM虚拟指令集，同时，在这个过程中可行做尽可能多的优化工作，以此来减少连接器的工作量和工作时间。例如：对于C/C++的前端可以将printf("hell\n"); 转换成puts("hello");。

总之，前端静态编译器可以做三件事：

1. 进行特定语言的优化

2. 将源程序翻译成LLVM代码

3. 在模块水平调用LLVM的passes进行全局变量和过程间优化

其中，1和3是选择性的，2是必须做的。

链接时：连接器和过程间优化器

连接时是一个做全程序主动过程间优化的天然时机，应该在该阶段是第一次将整个程序的各个部分整合到一起。连接时的优化是在IR上完成的，同时它还利用了高层信息来使得优化更加高效。在此期间，编译器做了大量的优化工作：

1. 数据结构分析（文本敏感的指向分析）

2. 调用图构建

3. Mod/Ref分析

4. 内联（inlining）

5. 无效全局变量、参数、类型消除

6. 常量替换（constant propagation）

7. 数组边界检测消除

8. 简单结构体域重排

9. 自动池分配

在LLVM设计中，这种编译时和连接时优化器还使用了一种很常用的技术来提高优化性能——在编译时计算函数的摘要信息，并且将其追加到LLVM字节码（可以看做IR的另一种表述）中，在连接时，连接时过程间的优化器将可以使用这些摘要信息而不是直接分析LLVM字节码，这样便使得优化工作更为简便。

在连接时一个重要的结果就是生成本地机器码，事实本地机器码生成细分下来有两种方式——离线生成和JIT本地代码生成。

离线生成

在程序执行之前，使用代码生成器将LLVM代码翻译成本地代码。这种代码生成器在连接时和安装时静态运行的，通常可以产生较高性能的可执行程序但是这种方式需要较高代价。如果用户使用的是后连接优化，那么LLVM代码必须和可执行程序打包在一起，并且程序中还需要插入轻量级的指令以识别循环和热点函数等，这无疑加大了编译难度。

JIT本地代码生成

JIT执行引擎技术：该技术可以在程序运行时调用合适的代码生成器每次翻译一个函数来执行。当然这种技术也可以像离线代码生成技术那样，在程序中插入相应的指令来收集对应信息。但这种逐条翻译执行的方案无疑会影响程序运行。

总之，两种方式各有利弊。

运行时：概要信息（profiling）和再优化

在传统编译器的设计策略中大多只考虑前两个阶段的优化，而在LLVM中引入了一种新的优化策略——运行时优化。这种策略通过收集运行时的概要信息并且利用这些收集来的信息指导对LLVM字节码的再优化和再编译。

概要信息的获取

正如在上文中提到的，传统编译器收集概要信息（如果使用这种优化策略的话，一般情况下开发者不愿意使用）是由开发者来完成，然后开发者再利用收集来的信息作为程序运行的反馈来做优化。但是LLVM运行时优化则不同：

1. 从最终用户那里收集运行时信息，而非开发者在测试过程中收集

2. 自动收集而非通过开发者以各种繁琐的方式收集

事实上，LLVM运行时优化器通过一系列的技术来收集信息从PC寄存器抽样技术，（来找出循环和热点函数）到路径概要技术（path profiling）（找出频繁被执行的路径）。

运行时优化方法

在程序运行的时候频繁被执行的执行路径通过离线和在线的指令被（如上文所述，本地机器代码生成器在生成本地机器代码是插入的）识别。例如：由本地代码生成器插入的离线指令可以识别代码中的频繁被执行的循环区域，进而识别频繁被执行的执行路径，一旦这种路径被识别，就可以从原始的LLVM代码中将其拷贝出来对其在优化生成本地代码，并将其存放到software-managed trace cache中。这次运行时优化策略的优点在于：

1. 本地代码可以通过复杂的算法提前生成高效的可执行代码。

2. 本地代码生成器和运行时优化器可以并行的工作，这样就为运行时优化器充分获得本地代码生成器的支持。

3. 运行时优化器可以利用高层信息来做更复杂的优化。

空闲时：离线优化器

有些程序并不适合运行时优化，这些程序主要有这些特点：代码量大、没有调用特别频繁的区块。由上文可知，运行时优化主要在于提高热点函数和循环的性能。

LLVM表示是永久保留的，所以科研在运行的空闲时利用目标机器和用户信息对应于程序再优化。一个离线的、空闲时的优化器具有以下几个优点：

1. 可以利用应用程序运行时获取的用户信息来做优化

2. 可以针对目标机器的详细特征来裁剪代码

3. 由于是离线执行，它可以做比运行时优化器更多的激进优化

LLVM中间表示

“代码表示”是LLVM区别于其他系统的一个主要特征。“代码表示”被设计为提供高层次程序信息以支持复杂分析和转换，同时，也足够底层来支持对任意代码的表示和静态编译。

LLVM指令集被设计为一种带有高层次类型信息的底层表示。即LLVM指令集（或者中间表示IR）有两个特点：底层表示和高层次信息。

指令集概述

LLVM虚拟指令集并不关心运行时和操作系统函数（如，I/O、内存管理、信号量等）。但是与传统编译器中间表示（如GCC的RTL）不同的是它是可读的，类似文本文件可以存在磁盘上的。LLVM虚拟指令集主要有以下几个特点：

1. LLVM指令集包括通用处理器的主要操作，但是不包括有特定限制的指令。（如：物理寄存器、管道、底层函数调用等）。

2. LLVM提供无限的指定类型的虚拟寄存器，可以存储基本类型数据。这些虚拟寄存器是SSA格式的。

3. LLVM通过load/store操作来完成虚拟寄存器和内存之间的数据交换的。

4. LLVM也明确的构建了每个函数的控制流图已经异常控制流。

5. LLVM指令集中只包括31条指令：1、避免多条操作指令对应一种操作；2、指令重载

LLVM利用SSA作为主代码表示，每个虚拟寄存器被写入一条指令，每当使用一个寄存器依他定义为主。内存分配不是利用SSA格式的，许多可行的分配，在某次存储的时候通过指针对其进行了改变，使得它无法构建一个合理的、紧凑的SSA格式。SSA格式提供了一个紧凑的def-use图简化了数据流优化并且使得流不敏感算法能够快速的、不经过复杂数据流分析的达到流敏感信息。非循环转换在SSA形式的进一步简化，因为他们不会遇到遇到依赖SSA寄存器的反或输出。非存储器的转换也大大简化，因为（无关SSA）的寄存器不能有别名。每个函数由一组基本块构成，每个基本快由一个指令序列构成。

语言无关的类型系统

语言无关类型系统是LLVM的基本特性。LLVM是一个严格的指定类型的表示。每一个SSA寄存器和明确的内存对象都有一个相应的类型，这个类型信息用于连接指令操作码和指令语法定义。它 提供一个语言独立的类型系统，包括四个简单类型：void、布尔、整数和浮点数， 以及四个导出类型：指针、数组、结构和函数。这个简单的类型系统可以实现绝大多数高级语言的类型，比如，C++中的类可以使用结构、函数以及函数指针的数 组的组合实现。

cast指令：类型转换指令。该指令是进行类型转换的唯一方法。换而言之，LLVM代码中类型转换都是显式的。

getelementptr指令：用来实现指针算术运算，它可以计算组合类型（结 构或者数组）数据的成员的地址。getelementptr使得在LLVM代码中附带类型信息成为可能。例如：X[i].a = 1; 将被转换成如下代码：

%p = getelementptr %xty\* %X, long %i, ubyte 3;

store int 1, int\* %p;

内存分配和一致性内存模型

一些程序之所以难以充分优化，关键在于在内存分配是在堆中大量使用复杂的数据结构。为了解决这个问题LLVM为指定类型内存分配提供了指令：

1. malloc指令：在堆上分配一个或多个特定类型单元，并且返回指向新开辟内存的指定类型的指针。

2. free指令：释放有malloc开辟的内存空间。

3. alloca指令:和malloc类似，只不过alloca在函数栈空间开辟内存空间，并且在函数结束时自动回收，无需使用free，同样也返回指定类型的指针。

这些指令的基本目的在于确保LLVM中间表示的类型安全，并且使得对数据结构分析和动态池分配（Automatic Pool Allocation）更加简单。另一个更为重要的属性就是，在转换类型不安全语言的代码也能够确保类型安全。

在LLVM系统中LLVM虚拟指令集是管理内存的唯一方法。在LLVM中，所有的可寻址对象都被精确额分配了内存空间。全局变量和函数定义了一个符号，该符号提供全局变量或函数的地址，而非其本身。这样便定义了一个统一的内存模型。所有的内存操作，包括调用指令都是通过指针完成的。没有隐含的访存操作，使得内存访问分析更加简化，而且代码表示也不再需要地址操作符。

函数调用和异常处理机制

LLVM提供了两种函数调用指令，这些指令抽象了底层机器相关的函数调用约定、简化了程序分析并且为异常处理提供支持。call 指令使用一个指针指向一个函数来进行函数调用。另外一个指令invoke 通常被用来做异常处理相关工作。

LLVM实现了一种栈展开机制，该机制使得异常处理“零代价”。该机制说明，但异常没有被抛出时程序无需执行额外的指令。相反，如果指令被抛出，栈将会被展开，直到函数调用的返回地址。LLVM运行时保存了一个静态的返回地址到异常处理块的映射，这样，当栈展开是就可调用异常处理器了。为了建立这种异常处理器信息的映射，invoke 指令是在call 指令追加了异常处理的标签。当异常产生时，invoke的返回地址和异常处理标签联系在一起，这样就可以执行异常处理代码了。

invoke 指令可以直接使用LLVM底层概念表示高级语言的异常处理语法，这样就使得LLVM表示独立于源代码的异常处理语法了。在这种表示中，异常被直接转换并且对LLVM框架是可见的，以此确保所有的关于异常处理的LLVM翻译都是正确的。

离线和内存内表示

LLVM的中间表示是一种first class language，和文本文件、可执行的二进制文件一样，可以保存到硬盘中。这也是为什么有些人会将LLVM中间表示直接翻译成LLVM汇编的原因。LLVM中间表示是可以可读的，如同汇编代码一样。其他编译器的中间表示大多是种内存中的复杂数据结构，以至于很难写出来，这让其他编译器既难懂又难以实现。

LLVM中间表示的这种特性使得其可以无损的、很简单的进行调试转换，测试用例的编写更加容易，同时也减少了理解内存内表示的时间。

**LLVM指令集**

数据类型:

包含源语言独立的预定义的数据类型：void, bool, signed/unsigned int，single/double float ->便于编写可移植代码

包含4种派生类型：pointer, array, structure, function -> 可表示复杂的独立于语言的分析优化中使用的类型信息

声明的类型信息可能不可靠，因为LLVM是语言独立的，必须能支持一些弱类型的语言。必须使用一些指针分析算法来区分指针目标类型明确但指针的源头类型不明确的内存访问

Cast: （唯一的）类型转换指令

Getelementptr：为底层代码保存类型信息的一个困难是实现地址算法。该指令给定一个用于聚合类型化对象的类型化指针, 它可以引用子元素的地址, 保存类型

函数调用和例外:

Call: 利用一个类型化的函数指针进行一般的函数调用

LLVM在高级语言中实现异常处理: 提供显式的、低级的、独立于机器的机制, 比如支持C语言中的setjmp, longjmp

Invoke/unwind: 支持抽象例外

invoke用于指定在堆栈展开过程中必须执行的异常处理代码。

unwind用来引发异常或执行 longjmp。

invoke指令的工作方式类似call, 但指定一个额外的基本块, 用来指示unwind处理程序的起始块。当程序执行unwind指令时, 它在逻辑上展开堆栈, 直到它删除由调用创建的激活记录。然后, 它将控制权转移到调用指定的基本块。

参考文献

[Introduction to the LLVM Compiler](http://llvm.org/pubs/2008-10-04-ACAT-LLVM-Intro.html) （LLVM编译器的介绍）  
Book chapter providing a compiler hacker’s introduction to LLVM.

<http://llvm.org/pubs/2002-12-LattnerMSThesis.pdf>

Chris Lattner Vikram Adve《LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation》

CHRIS ARTHUR LATTNER 《LLVM: AN INFRASTRUCTURE FOR MULTI-STAGE OPTIMIZATION》

《KLEE软件分析》

https://github.com/chyyuu/symexe/blob/master/klee\_document/klee\_analysis.md

《为什么人人都该懂点LLVM》<http://geek.csdn.net/news/detail/37785>

[LLVM: An Infrastructure for Multi-Stage Optimization](http://llvm.org/pubs/2002-12-LattnerMSThesis.html)  (LLVM: 一个多级优化的基础设施)