نگاهی ژرف به طراحی، ساختار و مزایای LLVM

بخش پایانی

» نویسنده: کریس لاتنر » منبع: دکتر دابز » ترجمه: احمد شریف‌پور

در قسمت پیشین این مقاله به بررسی ویژگی‌های متفاوت و منحصر به‌فرد LLVM پرداختیم و نتیجه ماجولار بودن این کامپایلر را در ساده‌سازی ایجاد Frontendهای مختلف برای زبان‌های برنامه‌نویسی متفاوت مشاهده کردیم. در این بخش به بررسی مزیات این طراحی ماجولار در قسمت بهینه‌سازی کد (Optimizer) و همین‌طور بخش مولد کد یا Backend کامپایلر خواهیم پرداخت و همین‌طور خواهیم دید که LLVM چگونه با استفاده از فایل‌های توصیف معماری مقصد، می‌تواند برای کامپایل برنامه‌ها روی گستره وسیعی از سخت‌افزارها به کار برده شود.

LLVM مجموعه‌ای از کتابخانه‌ها است

بعد از موضوعات مرتبط با طراحی LLVM IR، مهم‌ترین جنبه آن این است که LLVM به صورت مجموعه‌ای از کتابخانه‌ها طراحی شده است. این درست برخلاف رویکردی است که در کامپایلر خط فرمان GCC یا در ماشین‌های مجازی JVM یا دات‌نت دیده می‌شود. پروژه LLVM در واقع یک زیرساخت است؛ مجموعه‌ای از فناوری‌های مفید کامپایلر است که می‌توانند برای حل مسایل و مشکلات خاص (مثلاً ساخت یک کامپایلر C یا یک Optimizer در پایپ‌لاین جلوه‌های ویژه) مورد استفاده قرار بگیرند. این موضوع در حالی که یکی از قدرتمندترین ویژگی‌های LLVM است، یکی از نکات طراحی آن است که بسیار کمتر از بقیه شناخته شده است.

بیایید به عنوان نمونه، به طراحی یک Optimizer نگاهی بیاندازیم. این Optimizer کدهای LLVM IR را خوانده، آن را کمی دستکاری کرده و یک کد LLVM IR جدید تولید می‌کند که به نظر می‌رسد سریع‌تر اجرا می‌شود. در LLVM نیز همانند بسیاری از کامپایلرهای دیگر، Optimizer به صورت پایپ‌لاینی از چندین دور (Pass) اجرای مجزای توابع بهینه‌سازی طراحی شده است که هریک به صورت جداگانه روی کد ورودی اجرا شده و امکان اعمال برخی تغییرات را دارند. بررسی‌های خطی (Inliner)، نسبت‌دهی دوباره عبارت‌ها (Expression Reassociation) و جابه‌جایی کدهای نامتغیر حلقه‌ها، نمونه‌هایی معمول از این دورهای بررسی کد هستند. بسته به سطح بهینه‌سازی موردنظر، تعداد متفاوتی از دورهای بهینه‌سازی مورد استفاده قرار می‌گیرد. برای مثال کامپایلر Clang، با سوئیچ O0- حتی یک دور بهینه‌سازی را هم اجرا نمی‌کند، اما با سوئیچ O3– تعداد دورهای مورد استفاده بهینه‌سازی به ۶۷ دور می‌رسد (با نسخه ۲/۸ LLVM). هر یک دور LLVM به صورت یک کلاس C++ نوشته شده است که به صورت غیرمستقیم از کلاس Pass مشتق می‌شود. بیشتر این دورها در یک فایل واحد با پسوند cpp. نوشته شده‌اند و زیرکلاس‌های آن‌ها که از Pass مشتق می‌شوند، در یک فضای نام (namespace) بدون اسم تعریف شده‌اند. این امر باعث می‌شود که این کلاس‌ها private بوده و تنها در فایل تعریف‌کننده‌شان قابل استفاده باشند. برای این‌که یک دور بهینه‌سازی مفید باشد، کدهایی در بیرون فایل آن هم باید بتوانند به آن دسترسی داشته باشند. به همین دلیل از هر فایل تنها یک تابع (برای ایجاد آن دور) export

می‌شـود. در فهرست ۱ نمونه‌ای ساده شده از یک دور را مشاهده می‌کنید که این توضیحات را آشکارتر می‌کند.

```
namespace {
 class Hello : public FunctionPass {
 public:
  // Print out the names of functions in the LLVM IR being optimized.
  virtual bool runOnFunction(Function &F) {
   cerr << "Hello: " << F.getName() << "\n";
   return false;
  }
 };
}

FunctionPass *createHelloPass() { return new Hello(); }
```

فهرست۱ نمونه‌ای از تعریف یک دور از بهینه‌سازی کد

همان‌گونه که گفته شد، Optimizer طراحی شده برای LLVM، دورهای متعدد و متفاوتی را فراهم می‌کند که همه آن‌ها در قالب یکسـانی نوشـته شـده‌اند. این دورها در قالب یک یا چند فایل با پسـوند o. ترکیب شـده و سـپس در مجموعه‌ای از کتابخانه‌های آرشیوی (فایل‌هایی با پسوند a. در یونیکس) ذخیره می‌شوند.

این کتابخانه‌ها تمام قابلیت‌های تحلیلی و تبدیلی را فراهم می‌آورد و این دورها کمترین وابستگی و ارتباط را با یکدیگر دارند. انتظار می‌رود که این دورها مستقل از یکدیگر کار کنند یا وابستگی‌هایشان به سایر دورها را (در صورت وجود) به صورت صریح بیان کنند. وقتی تعداد دورهای موردنظر برای اجرا مشخص شده باشد، PassManager در LLVM از اطلاعات صریح وابستگی‌ها برای برآوردن نیازهای هر یک از دورها استفاده کرده و اجرای آن‌ها را بهینه می‌کند.

کتابخانه‌ها و قابلیت‌های تجریدی و انتزاعی بسیار عالی هستند، اما در واقع به حل مشکلات کمک نمی‌کنند. نکته جالب توجه زمانی آشکار می‌شود که کسـی بخواهد ابزار جدیدی تولید کند که از فناوری کامپایلر اسـتفاده می‌کند. مثلاً شـاید کسـی قصد سـاخت یک کامپایلر JIT را برای یک زبان پردازش تصویر داشته باشد.

سـازنده چنیـن کامپایلـری محدودیت‌هایـی را در ذهن خـود دارد. بـه عنوان مثال، شـاید ایـن زبان پردازش تصویر، به شـدت نسـبت به تأخیرهای زمان کامپایل حسـاس باشـد یا ویژگی‌هایی در این زبان وجود داشـته باشـد که بهینه‌سـازی آن‌ها برای افزایش کارایی از اهمیت زیادی برخوردار باشـد. طراحی کتابخانه محـور Optimizer در کامپایلر LLVM، این امکان را برای سـازنده این زبـان فراهم‌می‌آورد که هم دورهای مورد نظـر برای پـردازش تصویر و هم ترتیب اجرای آن‌هـا را انتخاب کند. اگر همه چیز به صورت یک تابع منفرد و بزرگ نوشـته شـده بود، صرف وقت برای استفاده دوباره از بخش‌های مختلف آن در طراحی کامپایلر این زبان پـردازش تصویر فرضی، چندان منطقی نبود. اگر تعداد اشـاره‌گرها اندک باشـد، تحلیل aliasها و بهینه‌سـازی حافظه ارزش چندانی نخواهد داشت. هرچند به‌رغم تمام تلاش‌های ما، LLVM نمی‌تواند به صورتی جادویی تمام مشکلات مربوط به بهینه‌سازی را حل کند.

چون در LLVM از زیرسیستم مربوط به دورهای بهینه‌سازی ماجولار است و خود PassManager چیزی درباره سازوکار درونی دورهای بهینه‌سازی نمی‌داند، برنامه‌نویسـان آزادند که دورهای خاص زبان موردنظر خود را طراحی کنند تا ضعف‌های موجود در دورهای بهینه‌سازی LLVM را پوشش داده یا فرصت انجام بهینه‌سـازی‌های خاص زبان موردنظرشان را داشته باشـند. شـکل ۱ نمونه ساده‌ای از سیسـتم پردازش تصویر فرضی ما را نشان می‌دهد.

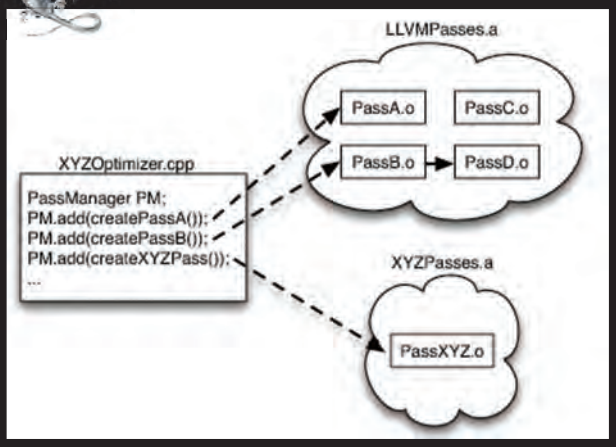
زمانـی کـه مجموعـه بهینه‌سـازی‌های موردنظر انتخاب شـدند (و تصمیمات مشـابه مربوط به مولد کد نیز گرفته شـدند)، کامپایلر مربوط به پردازش تصویر به صورت یک فایل اجرایی یا یک کتابخانه داینامیک ایجاد می‌شود. با توجه به این‌که تنها ارجاع به دورهای بهینه‌سازی LLVM، توابع سـاده‌ای هسـتند که به فایل‌های با پسوند o. اشـاره می‌کنند و با توجه به این‌که بهینه‌سـازها در فایل‌هایی با پسوند a. ذخیره شده‌اند، تنها دورهایی از بهینه‌سـازی که «مورد اسـتفاده قرار گرفته‌اند» به برنامه نهایی متصل خواهد شـد و نه کل Optimizer طراحی شـده بـرای LLVM. در مثال قبلی ما (شـکل ۱) به دلیل ارجاع برنامه بـه دورهای PassA و PassB این دورها به برنامه نهایی لینک می‌شـوند. با توجه به این‌که دور PassB برای انجام برخی تحلیل‌ها از دور PassD هم استفاده می‌کند، PassD هم لینک می‌شود. اما به عنوان مثال دور PassC (و بسیاری دورهای بهینه‌سازی دیگر) مورد اسـتفاده قرار نگرفته اسـت و به همین دلیل کدهای آن به برنامه پردازش تصویر فرضی ما لینک نمی‌شود.

ایـن همان جایی اسـت که طراحی کتابخانه محـور LLVM قدرت خود را بـه نمایش می‌گذارد. ایـن رویکرد طراحی سرراسـت، به LLVM اجازه می‌دهـد کـه قابلیت‌های بسـیار متنوعی را عرضه کند کـه برخی از آن‌ها تنهـا بـرای مخاطبانی خاص کاربرد خواهد داشـت و این قابلیت‌ها موجب دردسر کسانی که تنها به قابلیت‌های ابتدایی کامپایلر احتیاج دارند، نخواهد شـد. در نقطه مقابل، Optimizerهای کامپایلرهای سـنتی به صورت حجم انبوهی از کدها یکپارچه با ارتباطات درونی ایجاد می‌شـوند که جدا کردن بخش یا زیرمجموعه خاصی از آن‌ها، درک کردن سیسـتم کارشـان و راه افتادن در استفاده از آن‌ها را بسیار دشوار می‌کند. به کمک LLVM می‌توانید سـازوکار هر یک از Optimizerها را جداگانه درک کنید؛ بدون این‌که نیاز باشـد نحوه عملکرد و کنار هم قرار گرفتن کل سیسـتم را بفهمید. همچنین این طراحی کتابخانه محور، دلیلی اسـت که نشـان می دهد چرا برخی افراد درک نادرستی از چیستی LLVM دارند.

کتابخانه‌های LLVM قابلیت‌های فراوانی دارند، اما آن‌ها به خودی خود هیچ کاری انجـام نمی‌دهند. این وظیفه طراح برنامه کلاینت این کتابخانه‌ها (مثلا کامپایلر Clang برای زبان C) است که تعیین کند برای بهترین استفاده، ایـن قطعـات چگونه باید در کنار هم قرار بگیرنـد. این لایه‌بندی محتاطانه، factoring و تمرکز روی قابلیت‌های زیرمجموعه‌ها باعث می‌شود که بتوان Optimizer پروژه LLVM را در محیط‌های متنوع و برای کاربردهای بسیار متفاوت مورد اسـتفاده قرار داد. همچنین باید به خاطر داشـت که صرف فراهم شـدن قابلیت‌های کامپایل JIT توسـط LLVM به این معنی نیسـت که همه کلاینت‌ها باید از LLVM استفاده کنند.

طراحی مولد کد LLVM برای هدف گرفتن ماشین‌های مختلف

مولد کد LLVM وظیفه تبدیل LLVM IR به کدهای زبان ماشـین معماری مـورد نظـر را بر عهده دارد. از یک سـو این مولد کـد باید بهترین کد زبان ماشـین ممکن را بـرای هر معماری مقصد دلخواهـی تولید کند (در حالت ایده‌آل هر مولد کد باید به‌صورت اختصاصی برای کدهای ماشـین مقصد نوشـته شـده باشد) و از سـوی دیگر، مولدهای کد برای تمام ماشین‌های مقصد باید مشـکلات مشـابهی را حل کنند. برای نمونه، تمام مولدهای کد



شکل ۱ سیستم پردازش تصویر فرضی XYZ با استفاده از LLVM

باید مقادیری را به رجیسترها نسبت دهند و اگرچه هر معماری مقصد فایل‌های رجیستر جداگانه و متفاوتی دارد، اما الگوریتم‌های استفاده شده برای معماری‌های مقصد متفاوت باید تا حد ممکن مشابه و قابل استفاده دوباره باشند.

با رویکردی مشابه آن‌چه در بخش Optimizer دیدیم، مولد کد LLVM نیز مشکل تولید کد را با تفکیک آن به چند دور (Pass) مختلف حل کرده و تعدادی از دورهای درونی را تعریف می‌کند که به صورت پیش‌فرض انجام می‌شوند. انتخاب دستورالعمل، اختصاص رجیسترهای حافظه، زمان‌بندی، بهینه‌سازی چیدمان کد و انتشار اسمبلی از جمله این دورها هستند.

به این ترتیب، برنامه‌نویس Backend این شانس را خواهد داشت که دورهای موردنظر خود را از میان دورهای پیش‌فرض انتخاب کند، آن‌ها را بازنویسی کرده یا بسته به نیاز دورهایی کاملا اختصاصی را برای معماری موردنظر خود پیاده‌سازی کند. به عنوان مثال، Backend مربوط به معماری x86 به دلیل کم بودن تعداد رجیسترهای این معماری، از یک سیستم زمان‌بندی برای کاهش فشار روی رجیسترها (Register-Pressure-Reducing) استفاده می‌کند، اما Backend مربوط به معماری PowerPC به دلیل زیاد بودن تعداد رجیسترها از یک سیستم زمان‌بندی برای بهینه‌سازی تأخیر استفاده می‌کند. همچنین Backend مربوط به معماری x86 از یک دور اختصاصی استفاده می‌کند که پشته ممیز شناور x87 را مدیریت می‌کند و Backend مربوط به معماری ARM از دوری اختصاصی استفاده می‌کند که Constant Pool Island را در مکان‌های موردنیاز در درون تابع‌ها قرار می‌دهد. این انعطاف‌پذیری امکان تولید کدهای عالی را برای برنامه‌نویسان Backend فراهم می‌کند، بدون این‌که نیازی به نوشتن یک مولد کد جدید برای معماری مقصد وجود داشته باشد.

فایل‌های توصیف مقصد در LLVM

رویکرد ترکیب و تطبیق (Mix and Match) به برنامه‌نویسان Backend امکان می‌دهد که مواردی را که برای معماری موردنظرشان لازم به نظر می‌رسد، انتخاب کنند و هنگام انتقال به معماری‌های متفاوت و جدید، امکان استفاده دوباره از بسیاری از کدهای نوشته شده را فراهم می‌آورد. این امر چالش دیگری را به وجود خواهد آورد.

هر بخش کد مشترک میان معماری‌های سخت‌افزاری مختلف باید بتواند با سیستمی عمومی و فراگیر، درباره خصوصیات و ویژگی‌های معماری مقصد اطلاعاتی کسب کند. برای مثال یک تخصیص‌دهنده رجیستر مشترک باید بداند که فایل رجیستر معماری مقصد کدام است و برای دستورالعمل‌ها و عملیات متناظری که روی رجیسترهای حافظه انجام می‌شود، چه محدودیت‌هایی وجود دارد.

راه‌حل LLVM برای این مشکل، ارائه مشخصات معماری مقصد در قالب زبانی اختصاصی و توصیفی است که در مجموعه‌ای از فایل‌های با پسوند td. ذخیره شده و توسط ابزار tblgen مورد پردازش قرار می‌گیرند. نمونه‌ای ساده‌شده از فرآیند ایجاد کد مقصد برای معماری x86 در شکل ۲ قابل مشاهده است. زیرسیستم‌های متفاوتی که توسط فایل‌های td. پشتیبانی می‌شوند، این امکان را برای توسعه‌دهندگان Backend فراهم می‌آورد که بخش‌های متنوعی از معماری سخت‌افزاری موردنظرشان را ایجاد و پیاده‌سازی کنند.

به عنوان مثال، Backend مربوط به معماری x86 یک کلاس رجیستر را تعریف می‌کند که وظیفه آن نگه‌داری از کلیه رجیسترهای ۳۲بیتی معماری x86 است. این کلاس را که GR32 (در فایل‌های td. تمام تعاریف مربوط به معماری مقصد با حروف بزرگ نوشته می‌شوند) نامیده می‌شود، می‌توانید در فهرست ۲ مشاهده کنید.

```
def GR32 : RegisterClass<[i32], 32,
 [EAX, ECX, EDX, ESI, EDI, EBX, EBP, ESP,
  R8D, R9D, R10D, R11D, R14D, R15D, R12D, R13D]> { … }
```

فهرست ۲ تعریف کلاس GR32 برای معماری x86

این تعریف مشخص می‌کند که رجیسترهای معرفی شده در این کلاس می‌توانند مقادیر عددی صحیح ۳۲بیتی (i32) را نگه‌داری کنند و ترجیحاً باید به صورت ۳۲ بیتی انطباق داده شوند. ۱۶ رجیستر موجود با نام معرفی شده‌اند (که توصیف آن‌ها در بخش دیگری از فایل‌های td. درج شده است) و اطلاعات دیگری نیز وجود دارد که ترتیب ترجیحی تخصیص این رجیسترها و سایر موارد مرتبط را توضیح می‌دهند. با در دسترس بودن این توصیف، دستورالعمل‌های خاص می‌توانند به آن مراجعه کرده و از آن به عنوان یک عملوند استفاده کنند. برای مثال دستورالعمل "complement a 32bit register" همانند فهرست ۳ خواهد بود.

```
let Constraints = "$src = $dst" in
def NOT32r : I<0xF7, MRM2r,
      (outs GR32:$dst), (ins GR32:$src),
      "not{l}\t$dst",
      [(set GR32:$dst, (not GR32:$src))]>;
.
```

فهرست ۳ پیاده‌سازی دستورالعمل complement a 32bit register

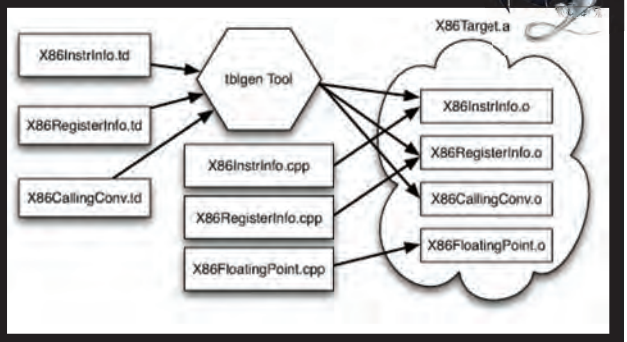
این توصیف تعیین می‌کند که NOT32r یک دستورالعمل است (از کلاس I tblgen استفاده می‌کند)، اطلاعات Encoding را فراهم کرده (0xF7, MRM2r)، مشخص می‌کند که قصد استفاده از $dst (یک رجیستر ۳۲بیتی خروجی) و $src (یک رجیستر ۳۲بیتی ورودی) را دارد. در این تعریف اشاره به کلاس GR32 رجیسترها تعیین می‌کند که استفاده از کدام رجیسترها مجاز خواهد بود. همچنین قواعد نحوی مورد استفاده برای اسمبلی این دستورالعمل (هم با قالب اینتلی و هم با قالب AT&T) را مشخص کرده، اثر این دستورالعمل و الگویی که باید در خط آخر با آن منطبق شود را تعریف می‌کند. کلمه کلید let در سطر نخست به تخصیص‌دهنده رجیسترها می‌گوید که رجیستر ورودی و خروجی باید به یک رجیستر فیزیکی واحد نسبت داده شوند.

این تعریف، توصیفی فشرده از دستورالعمل است و کدهای معمول LLVM می‌توانند به کمک tblgen اطلاعات بسیار زیادی را از آن استخراج کنند. کامپایلر برای انتخاب دستورالعمل مناسب براساس تطابق الگوها در کد ورودی IR، تنها به همین توصیف نیاز دارد. این توصیف به تخصیص‌دهنده رجیسترها می‌گوید که چگونه کارها را پردازش کند و برای تبدیل کردن دستورالعمل‌ها به کدهای زبان ماشین (و بالعکس) کافی خواهد بود. همچنین برای parse کردن و درج این دستورالعمل‌ها به صورت متنی نیز کفایت خواهد کرد. این قابلیت‌ها امکان پشتیبانی از ایجاد یک اسمبلر مستقل و منفرد برای معماری x86 را فراهم می‌کند. این اسمبلر به سادگی می‌تواند جایگزین gas (سرنام GNU assembler) شود. علاوه بر این، ایجاد دیس‌اسمبلر و حتی مدیریت دستورالعمل‌ها برای کامپایلرهای JIT نیز ازطریق این توصیف‌ها ممکن خواهد بود.

در کنار فراهم‌آوردن ویژگی‌های مفید، در اختیار داشتن قابلیت تولید اطلاعات مختلف از یک «حقیقت» واحد به دلایل دیگری نیز مفید خواهد بود. چنین رویکردی بروز تعارض میان اسمبلر و دیس‌اسمبلر بر سر قواعد نحوی (syntax) یا کدگذاری باینری را تقریباً غیرممکن می‌کند. همچنین توصیف‌های معماری مقصد را کاملاً آزمون‌پذیر می‌سازد. کدگذاری‌های دستورالعمل‌ها را می‌توان با Unit Testing و آن هم بدون درگیر کردن کل مولد کد کنترل کرد. با این‌که تلاش بر این است که بیشترین اطلاعات ممکن از یک معماری مقصد خاص در قالبی توصیفی در فایل‌های td. آورده شود، اما در هر صورت ما همه اطلاعات را در اختیار نخواهیم داشت. در عوض نویسندگان Backend باید برای برخی روال‌های پشتیبانی به نوشتن کدهای C++ پرداخته و دورهای اختصاصی معماری موردنظرشان را پیاده‌سازی کنند. هرچه تعداد معماری‌های مقصد پشتیبانی شده توسط LLVM بیشتر می‌شود، افزایش میزان اطلاعات توصیفی از معماری سخت‌افزاری که در فایل‌های td. گنجانده می‌شوند، اهمیت بیشتری می‌یابد. ما برای برطرف کردن این مشکل، در حال افزایش قابلیت‌های توصیفی فایل‌های td. هستیم. مزیت این امر این است که نوشتن Backend برای معماری‌های متفاوت به‌مرور زمان ساده‌تر و ساده‌تر می‌شود.

ویژگی‌های جذابی که از طراحی ماجولار نشأت می‌گیرند

ماجولار بودن علاوه‌بر این‌که ایده‌ای عالی در طراحی به شمار می‌رود، کتابخانه‌های فراوانی را با ویژگی‌های جذاب در اختیار کلاینت‌های LLVM قرار می‌دهد. این قابلیت‌ها از این موضوع نشات می‌گیرند که LLVM تنها برخی عملکردها را فراهم می‌کند اما تنظیم و تعیین بیشتر سیاست‌ها در زمینه استفاده از این عملکردها را بر عهده کلاینت می‌گذارد. همان‌طور که پیش‌تر نیز اشاره شد، کدهای LLVM IR را می‌توان به کمک سریال‌سازی



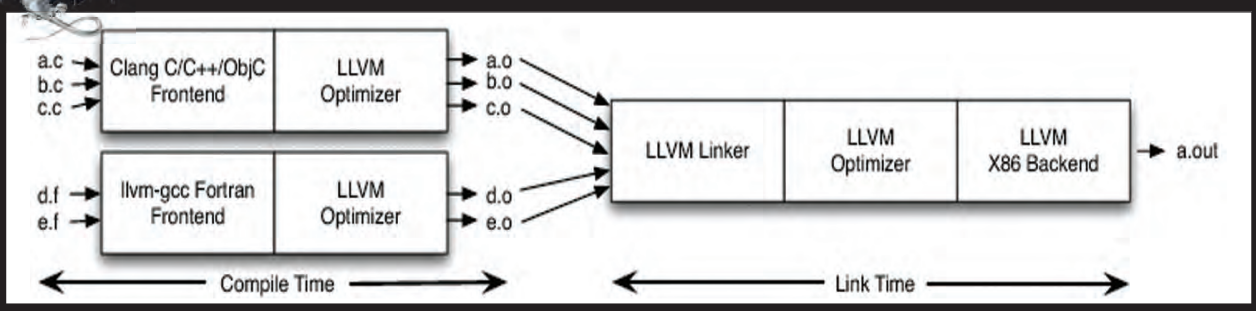
شکل ۲ توصیف ساده‌شده معماری x86

(serialization) با بازدهی بالا به قالبی باینری به نام بیت‌کدهای LLVM تبدیل کرد. عکس این کار نیز به آسانی امکان‌پذیر است. با توجه به این‌که LLVM IR تمام اطلاعات را در خود دارد و فرآیند سریال‌سازی نیز باعث از دست رفتن اطلاعات نمی‌شود، می‌توان بخشی از عملیات کامپایل را به انجام رسانده، نتیجه را روی دیسک ذخیره کرده و ادامه کار را در آینده از سر گرفت. این ویژگی تعدادی قابلیت جذاب را نیز به ارمغان می‌آورد که پشتیبانی از بهینه‌سازی در هنگام لینک کردن (Link-Time Optimization) و بهینه‌سازی در هنگام نصب‌کردن (Install-Time Optimization) از جمله آن‌ها هستند.

هر دوی این ویژگی‌ها نیازمند به تعویق افتادن تولید کد از زمان کامپایل هستند. اگر کامپایلری به صورت سنتی تنها توانایی دیدن تنها یک واحد ترجمه (مثلاً یک فایل با پسوند c. و تمام سرآیندهای موردنیازش) در هر زمان را داشته باشد، دیگر قابلیت بهینه‌سازی کل فایل‌های یک پروژه در یک قالب واحد و در تطبیق با یکدیگر را نخواهد داشت. بهینه‌سازی در هنگام لینک کردن یا LTO می‌تواند این مشکل را حل کند. کامپایلرهای مبتنی بر LLVM نظیر Clang می‌توانند با سوئیچ‌های خط فرمان flto– و O4–از این قابلیت استفاده کنند.

با استفاده از این سوئیچ‌ها، کامپایلر به جای نوشتن فایل‌های شیئی محلی (Native Object File) بیت‌کدهای LLVM را در فایل‌های o. ذخیره می‌کند و تولید کد مقصد را به زمان لینک کردن موکول می‌کند. این موضوع را می‌توانید در شکل ۳ مشاهده کنید.

جزئیات بر اساس سیستم‌عامل مورد استفاده ممکن است متفاوت باشند، اما نکته مهم این است که برنامه linker درک می‌کند که در فایل‌های o. به جای اشیای محلی، بیت‌کدهای LLVM ذخیره شده‌اند. در صورت مواجه شدن با چنین وضعیتی، linker محتوای تمام فایل‌ها را به حافظه منتقل



شکل ۳ بهینه‌سازی در هنگام لینک کردن یا LTO

کرده، آن‌ها را به هم لینک کرده و در نهایت بهینه‌ساز LLVM را روی کل کد حاصل اجرا می‌کند. چون در این وضعیت Optimizer بخش بزرگ‌تری از کد را می‌بیند، بهینه‌سازی‌های کامل‌تری (نظیر حذف کدهای زاید، inline کردن، استفاده از مقادیر ثابت بیشتر) را می‌تواند به انجام برساند. اگرچه بیشتر کامپایلرهای مدرن از LTO پشتیبانی می‌کنند، بیشتر آن‌ها (از جمله GCC، Open64 و کامپایلر اینتل) این کار را با یک فرآیند سنگین و کند سریال‌سازی انجام می‌دهند.

در LLVM قابلیت LTO به‌صورت مستقیم از طراحی سیستم نشات می‌گیرد و (برخلاف بسیاری از کامپایلرهای دیگر) روی گستره وسیعی از زبان‌های برنامه‌نویسی کار خواهد کرد؛ چراکه IR واقعاً مستقل از زبان برنامه‌نویسی مبدأ است. ایده بهینه‌سازی در هنگام نصب، تولید کد را حتی از هنگام لینک کردن نیز عقب‌تر برده و آن را تا زمان نصب برنامه به تأخیر می‌اندازد. نحوه کار این سیستم را می‌توانید در شکل ۴ ببینید. زمان نصب، به خصوص هنگامی که برنامه شما به صورت یک پکیج عرضه شود یا روی یک دستگاه قابل حمل دانلود یا آپلود شود، زمان بسیار جذابی است. زمان نصب هنگامی است که شما تازه مشخصات دستگاه مقصد را می‌یابید.

مثلاً در سخت‌افزارهای خانواده x86 تنوع فراوانی از تراشه‌ها و خصوصیت‌های سخت‌افزاری وجود خواهد داشت. با به تعویق انداختن انتخاب دستورالعمل، زمان‌بندی و سایر جنبه‌های تولید کد، می‌توان بهترین حالت را برای سخت‌افزاری که قرار است برنامه را اجرا کند، انتخاب کرد.

کامپایلرها بسیار پیچیده هستند و در آن‌ها کیفیت بسیار مهم است، به همین دلیل آزمایش آن‌ها امری حیاتی است. به عنوان مثال، پس از برطرف کردن اشکالی که باعث از کار افتادن Optimizer می‌شده است، یک آزمون رگرسیون نیز باید انجام شود تا اطمینان حاصل شود که این خطا دیگر رخ نخواهد داد. رویکرد سنتی آزمایش به این شکل است که فایلی مثلا با پسوند c. نوشته شده و به کامپایلر داده شود. سپس با سیستمی کنترل شود که آیا کامپایلر دچار اشکال خواهد شد یا خیر. این روشی است که برای کامپایلر GCC مورد استفاده قرار می‌گیرد. مشکل چنین رویکردی این است که کامپایلر از زیرسیستم‌های مختلفی تشکیل شده است و تعداد دورهایی که توسط Optimizer انجام می‌شود نیز بسیار زیاد هستند. در مثال قبلی، تمام این موارد ممکن است منبع بروز مشکل باشند یا پس از رفع باگ، دچار مشکلی جدید شوند. اگر چیزی در Frontend یا در Optimizer اولیه تغییر کند، یافتن منبع مشکل و نوشتن برنامه آزمون برای اطمینان از تصحیح کامل خطا به شدت دشوار خواهدبود.

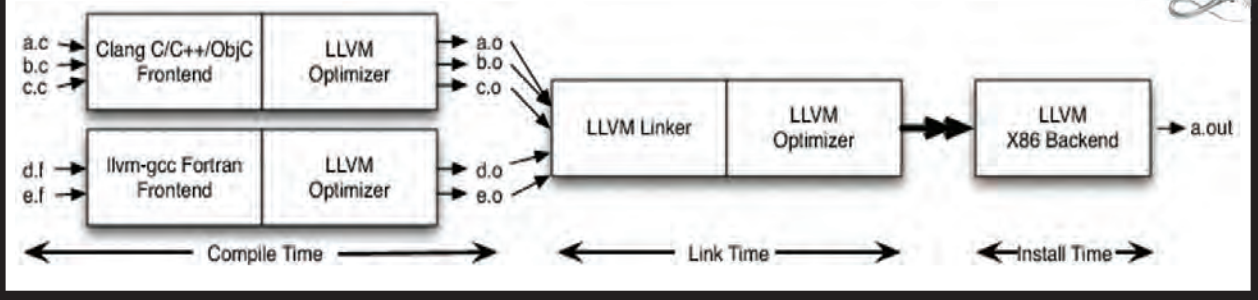
با استفاده از قالب متنی LLVM IR و یک Optimizer ماجولار، ابزارهای آزمودن LLVM شامل آزمون‌های رگرسیون هدف‌مندی است که می‌توانند LLVM IR را از روی دیسک خوانده، تنها یک دور از بهینه‌سازی را روی آن اجرا کرده و رفتار مورد انتظار از کامپایلر را بررسی کنند. فراتر از آزمون‌های مربوط به از کار افتادن کامپایلر، ممکن است یک آزمون بخواهد کنترل کند که آیا عملیات بهینه‌سازی واقعاً به انجام می‌رسد یا نه. در فهرست ۴ می‌توانید نمونه‌ای از این آزمون‌ها را ببینید که انجام دور بهینه‌سازی propagation روی دستورالعمل add را کنترل می‌کند.

```
; RUN: opt < %s -constprop -S | FileCheck %s
define i32 @test() {
 %A = add i32 4, 5
 ret i32 %A
 ; CHECK: @test()
 ; CHECK: ret i32 9
}
```

فهرست ۴ کنترل اجرای بهینه‌سازی propagation روی دستورالعمل add

خطی که با RUN شروع می‌شود، مشخص کننده دستوری است که باید اجرا شود. در این مورد دستورهای opt و FileCheck باید اجرا شوند. برنامه opt در واقع یک wrapper است که مدیر دورهای LLVM (سرنام LLVM Pass Manager) را در خود دارد و تمام دورهای استاندارد را لینک کرده و حتی امکان بارگذاری داینامیک پلاگین‌ها و سایر دورهای اختصاصی را داشته و آن‌ها را از طریق خط فرمان در اختیار کاربر قرار می‌دهد. ابزار FileCheck کنترل می‌کند که آیا ورودی استانداردش با یک سری از CHECKها مطابقت دارد یا خیر. در این مورد، آزمون ساده ما کنترل می‌کند که آیا دور constprop دستورالعمل جمع دو عدد ۴ و ۵ را با عدد ثابت ۹ جایگزین می‌کند یا خیر.

اگرچه این آزمون ممکن است مثالی ابتدایی و کم اهمیت به نظر برسد، انجام همین کار از طریق نوشتن یک فایل c. بسیار دشوار خواهد بود. بخش‌های Frontend معمولاً این کار را در هنگام parse کردن کد انجام می‌دهند و به همین دلیل نوشتن کدی که بتواند از این مرحله گذر کرده و انجام این کار را در یکی از دورهای بهینه‌سازی کنترل کند، بسیار دشوار و حتی ناممکن خواهد بود. اما چون LLVM IR را می‌توان به صورت یک فایل متنی بارگذاری کرده، دورهای دلخواه بهینه‌سازی را روی آن به انجام رساند و نتیجه را به صورت یک فایل متنی دیگر دریافت کرد، آزمودن آن‌چه مورد نظر است بسیار سرراست و راحت خواهد بود. خواه در هنگام آزمودن ویژگی‌های جدید باشد یا در هنگام انجام آزمون‌های رگرسیون.



شکل ۴ بهینه‌سازی در هنگام نصب‌کردن یا ITO

زمانی که یک باگ در کامپایلر یا سایر کلاینت‌های LLVM مشاهده می‌شود، قدم اول برای اصلاح آن ایجاد یک نمونه آزمون موردی است که بتواند این خطا را بازتولید کند. زمانی که این نمونه آزمایشی ایجاد شد، بهتر است آن را به ساده‌ترین شکل ممکن درآورد و تا حد ممکن به دقت محل بروز خطا (مثلاً یک دور از بهینه‌سازی که باعث ایجاد خطا می‌شود) را کشف کرد. اگرچه شما به تدریج یاد خواهید گرفت که چگونه این کار را به انجام برسانید، اما انجام این کارها زمانی که کامپایلر کدهای غلطی تولید می‌کند، بسیار دشوارتر و زمان‌برتر از زمانی است که کامپایلر ازکارافتاده و متوقف می‌شود.

ابزار BugPoint با استفاده از سریال‌سازی IR و ماجولار بودن طراحی LLVM، این فرآیند را به صورت خودکار به انجام می‌رساند. به عنوان مثال با مشخص کردن یک فایل ورودی با پسوند ll. یا bc. و همین‌طور تعدادی از دورهای بهینه‌سازی که باعث از کار افتادن کامپایلر می‌شود، ابزار BugPoint ورودی را به یک مجموعه دستورالعمل کوچک محدود کرده و دوری از بهینه‌سازی که عامل بروز مشکل است را دقیقاً مشخص می‌کند. خروجی این ابزار، آزمون خلاصه‌شده و دستور opt موردنیاز برای بازتولید خطا خواهد بود. این ابزار عملیاتش را با استفاده از تکنیک‌هایی شبیه دیباگ دلتا (Delta Debugging) به انجام می‌رساند. با توجه به این‌که BugPoint ساختار LLVM IR را درک می‌کند، برخلاف دستور معمول delta، هیچ زمانی را برای ایجاد کدهای IR نامعتبر و آزمودن آن با Optimizer تلف نمی‌کند.

در یک مورد پیچیده‌تر که با خطای کامپایل روبرو می‌شود، می‌توان ورودی، اطلاعات مولد کد، دستوری که باید به فایل اجرایی منتقل شود و یک ارجاع برای خروجی را تعیین کرد. در این حالت BugPoint ابتدا تعیین می‌کند که خطا به Optimizer مربوط است یا به مولد کد. پس از آن به‌صورت متناوب کد ورودی را به دو بخش تقسیم می‌کند. یک بخش «سالم» و یک بخش «مشکل‌دار». با انجام دادن این کار به صورت متناوب، BugPoint به تدریج قسمت مشکل‌دار را کوچک‌تر و کوچک‌تر می‌کند تا بتواند به حداقل کد آزمون دست یابد.

ابزار BugPoint ابزاری بسیار ساده است که به کمک LLVM توانسته است هزاران ساعت زمان را در فرآیندهای عیب‌یابی و تصحیح خطا صرفه‌جویی کند. هیچ کامپایلر اپن‌سورس دیگری چنین ابزار سودمندی را در اختیار ندارد، چرا که این ابزار به شدت به آن نحوه نمایش میانی (IR) تعریف‌شده توسط LLVM وابسته است.

با همه این‌ها BugPoint هم کامل نیست و بازنویسی آن می‌تواند منافع زیادی در بر داشته باشد. تاریخ نوشته شدن این ابزار به سال ۲۰۰۲ باز می‌گردد و معمولاً تنها هنگامی به روز شده است که کسی با ایراد و خطایی روبه‌رو شده که با ابزارهای موجود قابل رفع و رجوع نبوده است. این ابزار بدون هیچ طراحی اولیه یا مالکی در طی زمان رشد کرده و ویژگی‌های جدیدی (نظیر دیباگ کردن JIT) را کسب کرده است. LLVM از ابتدا برای دسترسی به قابلیت‌هایی که تاکنون شرح داده شد، ماجولار طراحی نشده است.

در ابتدا این موضوع مکانیسمی دفاعی بود چرا که مشخص بود ما نمی‌توانیم از ابتدا همه چیز را به صورت مرتب و صحیح پیاده‌سازی کنیم. مثلاً پایپ‌لاین دورهای بهینه‌سازی ماجولار به این دلیل ایجاد شدند که بتوانیم هر یک از دورها را از بقیه مجزا کنیم و به این ترتیب با ایجاد شدن هر پیاده‌سازی جدید از یکی از دورها، نسخه قدیمی را به راحتی کنار بگذاریم. من در بسیاری موارد گفته‌ام که هیچ یک از زیرسیستم‌های LLVM کاملاً خوب و بی‌نقص نخواهند بود، مگر این‌که حداقل یک‌بار از نو نوشته شوند.

یکی دیگر از جنبه‌های مهم و زیرکانه LLVM (که در میان کلاینت‌های کتابخانه‌های LLVM بسیار بحث‌برانگیز است)، تلاش ما برای بازنگری در تصمیمات قدیمی و ایجاد تغییرات گسترده در APIهای برنامه، بدون در نظر گرفتن نگرانی‌های مربوط به سازگاری با نسخه‌های گذشته (Backward Compatibility) است.

برای مثال ایجاد تغییرات گسترده در خود LLVM IR نیازمند به‌روز رسانی تمام دورهای بهینه‌سازی و در نتیجه اعمال تغییرات بسیار زیاد در APIهای C++ است. ما در برخی شرایط چنین کارهایی را انجام داده‌ایم و این کار به رغم دردسری که برای کلاینت‌ها به وجود می‌آورد، برای حفظ این روند سریع رشد و بهبود LLVM الزامی است. برای ساده‌تر کردن کار کلاینت‌های بیرونی و همین‌طور پشتیبانی از امکان اتصال به سایر زبان‌های برنامه‌نویسی، ما wrapperهای فراوانی را به زبان C برای بسیاری از APIها فراهم کرده‌ایم که به شدت پایدار هستند.

همچنین نسخه‌های جدید LLVM به خواندن فایل‌های قدیمی ll. و bc. ادامه خواهند داد. با نگاه به آینده ما قصد داریم که LLVM را ماجولارتر کرده و امکان استفاده از زیرمجموعه‌های آن را ساده‌تر کنیم. مثلاً قسمت مولد کد هنوز تا حد زیادی یکپارچه است. در حال حاضر، نمی‌توان برخی زیرمجموعه‌های LLVM را براساس ویژگی‌های موردنظر ایجاد کرده و مورد استفاده قرار داد. به عنوان مثال، اگر شما بخواهید از JIT استفاده کنید اما به اسمبلی inline، مدیریت استثنا یا تولید اطلاعات دیباگ احتیاج نداشته باشید، باید بتوانید مولد کد را بدون تمام این قابلیت‌ها مورد استفاده قرار دهید (کاری که اکنون امکان‌پذیر نیست).

ما همچنین به صورت پیوسته کیفیت کد تولید شده توسط Optimizer و مولد کد را بهبود داده‌ایم، قابلیت‌هایی را به IR افزوده‌ایم تا از زبان‌های جدید و معماری‌های مقصد جدید، بهتر پشتیبانی کند و امکان انجام بهینه‌سازی‌های سطح بالای مختص زبان‌های برنامه‌نویسی خاص در LLVM فراهم شود.

پروژه LLVM به رشد خود ادامه خواهد داد و از جهات مختلف بهبود خواهد یافت. مشاهده روش‌های متفاوت استفاده از LLVM در سایر پروژه‌ها و همین‌طور این‌که در محیط‌های مختلف و جدیدی (که حتی به ذهن سازندگانش هم نمی‌رسید) کاربرد می‌یابد، بسیار خوشایند است. دیباگر جدید LLDB نمونه‌ای بارز از این موضوع است. این دیباگر از parserهای C++، C یا Objective-C موجود در Clang استفاده می‌کند تا بتواند عبارات را parse کند و از JIT موجود در LLVM برای ترجمه آن‌ها به کدهای مقصد استفاده می‌کند. سپس با استفاده از دیس‌اسمبلرهای LLVM و فایل‌های توصیف معماری مقصد به مدیریت نحوه انجام فراخوانی‌ها و امور دیگر می‌پردازد. امکان استفاده دوباره از کدهای موجود این پروژه، این فرصت را برای توسعه‌دهندگان دیباگرها فراهم می‌آورد تا تمرکز خود را به منطق دیباگر معطوف کنند نه این‌که دوباره گرفتار بازنویسی یک parser به زبان C++ شوند.

به‌رغم موفقیت‌هایی که LLVM تاکنون داشته است، هنوز کارهای زیادی برای انجام باقی مانده است. هنوز این خطر هم وجود دارد که با گذشت زمان LLVM چابکی خود را از دست داده و گرفتار تحجر و خمودگی شود؛ هرچند هیچ راه‌حل جادویی برای این مشکل وجود ندارد. اما من امید دارم که قرار گرفتن مداوم در معرض چالش‌ها و مشکلات در حوزه‌های تازه، اعتقاد به بازنگری در تصمیم‌های قدیمی، طراحی مجدد و دور ریختن کدهای قدیمی، به حفظ این چابکی کمک کند. به هر حال هدف ما کامل بودن صددرصد نیست، بلکه بهتر شدن در گذر زمان است.