LLVM リンク時の最適化設計と実装

このドキュメントは、2022 年 9 月 26 日時点で llvm.org に公開されている「LLVM Link Time Optimization: Design and Implementation」の日本語参考訳です。原文は更新される可能性があります。原文と翻訳文の内容が異なる場合は原文を優先してください。

説明

LLVM は、リンク時の最適化 (LTO) を可能にする強力なモジュール間の最適化機能を備えています。リンク時の最適化は、リンクステージで行われるモジュール間の最適化の別名です。このドキュメントは、LTO オプティマイザーとリンカー間のインターフェイスと設計について説明します。

設計思想

LLVM の LTO オプティマイザーは、コンパイラー・ツールチェーンでモジュール間の最適化を行いながら完全な透過性を実現しています。これは、開発者が `makefile` やビルドシステムを大幅に変更することなく、モジュール間の最適化の恩恵を受けることを可能にし、リンカーとの密な連携により実現されています。このモデルでは、リンカーは LLVM ビットコード・ファイルをネイティブのオブジェクト・ファイルのように扱い、それらをミックスおよびマッチングすることができます。リンカーは、LLVM ビットコード・ファイルを扱うため、共有オブジェクト・ファイルである libLTO を使用します。リンカーと LLVM オプティマイザーの密な統合により、他のモデルでは実現できない最適化が可能になります。リンカーからの情報により、オプティマイザーは保守的なエスケープ解析への依存を回避できます。

リンク時の最適化の例

次の例は、LTO による統合的なアプローチとクリーンなインターフェイスの利点を示しています。この例を実証するには、本ドキュメントで説明するインターフェイスで LTO をサポートするリンカーが必要です。ここでは、clang は透過的にシステムリンカーを呼び出しています。

- 入力ソースファイル `a.c` を LLVM のビットコード形式にコンパイルします。
- 入力ソースファイル `main.c` をネイティブ・オブジェクト・コードにコンパイルします。

```
--- a.h ---
extern int foo1(void);
extern void foo2(void);
extern void foo4(void);

--- a.c ---
#include "a.h"

static signed int i = 0;

void foo2(void) {
  i = -1;
}

static int foo3() {
  foo4();
  return 10;
}

int foo1(void) {
  int data = 0;

  if (i < 0)
    data = foo3();

  data = data + 42;
  return data;
}

--- main.c ---
```

```c
#include <stdio.h>
#include "a.h"

void foo4(void) {
  printf("Hi¥n");
}

int main() {
  return foo1();
}
```

コンパイルと実行

```
% clang -flto -c a.c -o a.o       # <-- a.o は LLVM ビットコード・ファイル
% clang -c main.c -o main.o       # <-- main.o はネイティブ・オブジェクト・ファイル
% clang -flto a.o main.o -o main   # <-- 標準のリンクコマンドに -flto を追加
```

- この例では、リンカーは `foo2()` が LLVM ビットコード・ファイルで定義される外部シンボルであることを認識しています。リンカーは通常のシンボル解決を完了して `foo2()` が未解決であることを検出します。この情報は LLVM オプティマイザーによって参照され `foo2()` を排除します。
- `foo2()` が排除されると同時に、オプティマイザーは `i < 0` 条件が常に偽であり、`foo3()` は参照されることがないことを認識します。これにより、オプティマイザーは `foo3()` も排除します。
- これは、リンカーが `foo4()` も排除できることを意味します。

この例は、リンカーと密な統合を行う利点を示しています。ここで、オプティマイザーはリンカーの介入なしで `foo3()` を排除することはできません。

代替方法

コンパイラー・ドライバーが LTO オプティマイザーを別途起動

このモデルでは、LTO オプティマイザーはリンカーが通常のシンボル解決で収集した情報を利用することができません。上記の例では、`foo2()` は外部参照であるため、リンカーからの情報がないとオプティマイザーは `foo2()` を排除できません。そのため、オプティマイザーは `foo3()` を排除できません。

すべてのオブジェクト・ファイルからシンボル情報を収集する別のツールを使用

このモデルでは、リンク時の最適化に必要な情報をリンカーではなく別のツールやライブラリーで収集します。このコードの重複は正当化が困難だけではなく、ほかにもデメリットが生じます。例えば、各種プラットフォー

ムのリンカーが提供するリンク時のセマンティクスや機能は一意ではありません。この新しいツールがそのようなすべての機能とプラットフォームを 1 つのツールでサポートするか、プラットフォームごとに個別のツールが必要になります。そのため、LTO オプティマイザーの保守コストが大幅に増加し現実的とは言えません。また、このアプローチは、LTO オプティマイザーの主眼ではないプラットフォーム・リンカーの開発と連携が必要となります。最後に、このアプローチでは、他のツールとリンカーで行う作業が重複するため、ビルド時間が長くなる傾向があります。

`libLTO` とリンカー間の複数フェーズ通信

リンカーは、さまざまなリンク・オブジェクトのシンボル定義とその使用に関する情報を収集します。これは、通常のビルドサイクルで他のツールが収集する情報よりも正確であるといえます。リンカーは、ネイティブの `.o` ファイル内のシンボル定義と利用を調査し、シンボルの可視性情報を介して情報を収集します。また、エクスポートされたシンボルリストなど、ユーザーが提供する情報も使用します。LLVM オプティマイザーは、制御フローとデータフローに関する情報を収集し、オプティマイザーの観点から見たプログラムの構造をより多く知ることができます。ここでの目標は、リンカーとオプティマイザーの密な統合を活用して、さまざまなリンクステージでこの情報を共有することです。

フェーズ 1: LLVM ビットコード・ファイルの読み取り

リンカーはすべてのオブジェクト・ファイルを順番に読み取り、シンボル情報を収集します。これには、LLVM ビットコード・ファイルだけではなく、ネイティブ・オブジェクト・ファイルも含まれます。すべての `.o` ファイルがネイティブ・オブジェクト・ファイルである場合のリンクコストを最小限にするため、リンカーは、供給されたオブジェクト・ファイルがネイティブ・オブジェクト・ファイルでないことが判明した場合にのみ `lto_module_create()` を呼び出します。ファイルが LLVM ビットコードであることを `lto_module_create()` が返すと、リンカーは `lto_module_get_symbol_name()` と `lto_module_get_symbol_attribute()` を使用して繰り返しモジュール全体で定義および参照されるすべてのシンボルを収集します。この情報は、リンカーのグローバル・シンボル・テーブルに追加されます。

`lto*` 関数は、すべての共有オブジェクト `libLTO` に実装されています。これにより、リンクツールとは独立して LLVM LTO コードを更新することができます。これをサポートするプラットフォームでは、共有オブジェクトは遅延ロードされます。

フェーズ 2: シンボル解決

このステージでは、リンカーはグローバル・シンボル・テーブルを参照してシンボルを解決します。未定義シンボルエラー、アーカイブメンバーの読み込み、弱いシンボルの置換などがレポートされます。リンカーは、与えられた LLVM ビットコード・ファイルの正確な内容を理解することなく、これをシームレスに行うことができます。デッドコード削除が有効である場合、リンカーはライブシンボルのリストを収集します。

フェーズ 3: ビットコード・ファイルの最適化

シンボル解決の後、リンカーはネイティブ・オブジェクト・ファイルに必要なシンボルを LTO 共有オブジェクトに通知します。上記の例では、リンカーは `lto_codegen_add_must_preserve_symbol()` を使用するネイティブ・オブジェクト・ファイルで `foo1()` だけが使用されることを報告します。次に、リンカーは `lto_codegen_compile()` を使用して LLVM オプティマイザーとコード・ジェネレーターを呼び出し、LLVM ビットコード・ファイルをマージしてさまざまな最適化パスを適用したネイティブ・オブジェクト・ファイルを返します。

フェーズ 4 : 最適化後のシンボル解決

このフェーズでは、リンカーはネイティブ・オブジェクト・ファイルを最適化して読み込み、変更があればそれらを反映して内部グローバル・シンボル・テーブルを更新します。また、リンカーは LLVM ビットコード・ファイルの外部シンボル参照状況の変更に関しても情報を収集します。上記の例では、リンカーは `foo4()` が使用されないことを指摘します。デッドコード削除が有効な場合、リンカーはライブシンボル情報を適切に更新して、デッドコードの削除を行います。

このステージ後、リンカーは LLVM ビットコード・ファイルを無視してリンクを続行します。

libLTO

`libLTO` は LLVM ツールの一部であり、リンカーが使用することを目的とした共有オブジェクトです。`libLTO` は LLVM 内部構造の詳細を公開することなく、LLVM のプロシージャー間のオプティマイザーが使用する抽象化された C インターフェイスを提供します。今後 LLVM オプティマイザーが進化しても、安定したインターフェイスを維持することを意図しています。全く異なるコンパイラー技術によって、オブジェクト・ファイルと標準リンカーで動作する `libLTO` を提供することも可能でしょう。

lto_module_t

非ネイティブ・オブジェクト・ファイルは、`lto_module_t` を介して処理できます。 次の関数を使用して、リンカーは (ディスクまたはメモリーバッファーの) ファイルが `libLTO` で処理可能なファイルであるか確認できます。

```
lto_module_is_object_file(const char*)
lto_module_is_object_file_for_target(const char*, const char*)
lto_module_is_object_file_in_memory(const void*, size_t)
lto_module_is_object_file_in_memory_for_target(const void*, size_t, const char*)
```

オブジェクト・ファイルが `libLTO` で処理できる場合、リンカーは次の関数のいずれかを使用して `lto_module_t` を作成します。

```
lto_module_create(const char*)
lto_module_create_from_memory(const void*, size_t)
```

完了したら、次の関数でハンドルを解放します。

```
lto_module_dispose(lto_module_t)
```

リンカーは、シンボル数と各シンボルの名前と属性を取得することで、非ネイティブ・オブジェクト・ファイルを調査できます。

```
lto_module_get_num_symbols(lto_module_t)
lto_module_get_symbol_name(lto_module_t, unsigned int)
lto_module_get_symbol_attribute(lto_module_t, unsigned int)
```

シンボルの属性には、アライメント、可視性、種類などがあります。

Darwin 上のオブジェクト・ファイルを扱うツール (`lipo` など) は、CPU タイプなどのプロパティーを識別するため、次の関数を使用します。

```
lto_module_get_macho_cputype(lto_module_t mod, unsigned int *out_cputype,
unsigned int *out_cpusubtype)
```

lto_code_gen_t

リンカーは非ネイティブ・オブジェクト・ファイルを `lto_module_t` にロードした後、`libLTO` にそれらを処理してネイティブ・オブジェクト・ファイルを生成することを要求します。これはいくつかのステップで行われます。最初に、以下の関数でコード・ジェネレーターを作成します。

```
lto_codegen_create()
```

次に、非ネイティブ・オブジェクト・ファイルをコード・ジェネレーターに追加するため、以下を使用します。

```
lto_codegen_add_module(lto_code_gen_t, lto_module_t)
```

リンカーは、いくつかの `codegen` オプションを設定できます。DWARF デバッグ情報を生成するかどうかは、以下の関数で設定します。

```
lto_codegen_set_debug_model(lto_code_gen_t)
```

位置情報の独立性は、以下の関数で設定します。

```
lto_codegen_set_pic_model(lto_code_gen_t)
```

ネイティブ・オブジェクト・ファイルから参照される、あるいは最適化の必要がないシンボルは、以下の関数で設定されます。

```
lto_codegen_add_must_preserve_symbol(lto_code_gen_t, const char*)
```

これらが設定された後、以下の関数を使用して、リンカーは設定されたモジュールからネイティブ・オブジェクト・ファイルを生成します。

```
lto_codegen_compile(lto_code_gen_t, size*)
```

この関数は、生成されたネイティブ・オブジェクト・ファイルを含むバッファーへのポインターを返します。リンカーは、ポインターが示す情報を解析し、残りのネイティブ・オブジェクト・ファイルとリンクを行います。