LTO Remove Dead Symbol

Link-time optimization (LTO)，顾名思义，编译器在链接时对程序执行的一种程序优化。对于像 C 这样语言，编译是逐个编译单元去编译的，然后通过链接器将这些编译单元链接在一起，LTO 就是在将这些编译单元链接在一起时执行的 intermodular optimization。

Remove Dead Symbol

在 LTO 阶段可以完成很多编译时无法做到的优化，如：在链接产物中删掉不会用到的死函数。

本文是对 LTO remove dead symbol 源码实现的阅读笔记（源码阅读基于 llvm 13.0.0 版本）。

Example

首先，我们举个例子，尝试一下 LTO remove dead symbol。

给定以下源文件：

```
--- tu1.c ---
int unused(int a);
int probably_inlined(int a);
int main(int argc, const char *argv[]) {
  return probably_inlined(argc);
}
```

```
--- tu2.c ---
int unused(int a) {
  return a + 1;
}
int probably_inlined(int a) {
  return a + 2;
}
```

编译 tu1.c 和 tu2.c 得到 tu1.o 和 tu2.o（通过选项 `-flto` 来开启 LTO）

```
% clang -flto -c tu1.c -o tu1.o
% clang -flto -c tu2.c -o tu2.o
```

链接 a.o 和 main.o 得到可执行文件 main（通过选项 `-fuse-ld=lld` 指定使用 lld linker)

```
% clang -flto -fuse-ld=lld tu1.o tu2.o -o main
```

可以通过 `readelf -sW ./main | awk '$4 == "FUNC"'` 查看生成的可执行文件的符号表中都有哪些函数：

```
% readelf -sW ./main | awk '$4 == "FUNC"'
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND
__libc_start_main@GLIBC_2.2.5 (2)
     3: 00000000002015e0     0 FUNC    LOCAL  DEFAULT   12
deregister_tm_clones
     4: 0000000000201610     0 FUNC    LOCAL  DEFAULT   12
register_tm_clones
     5: 0000000000201650     0 FUNC    LOCAL  DEFAULT   12
__do_global_dtors_aux
     8: 0000000000201680     0 FUNC    LOCAL  DEFAULT   12 frame_dummy
    15: 0000000000201740    15 FUNC    LOCAL  DEFAULT   12 probably_inlined
    16: 00000000002015d0     2 FUNC    LOCAL  HIDDEN    12
_dl_relocate_static_pie
    21: 0000000000201700     2 FUNC    GLOBAL DEFAULT   12 __libc_csu_fini
    22: 00000000002015a0    43 FUNC    GLOBAL DEFAULT   12 _start
    23: 0000000000201690   101 FUNC    GLOBAL DEFAULT   12 __libc_csu_init
    24: 0000000000201710    36 FUNC    GLOBAL DEFAULT   12 main
    27: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND
__libc_start_main
    30: 0000000000201750     0 FUNC    GLOBAL DEFAULT   13 _init
    31: 0000000000201768     0 FUNC    GLOBAL DEFAULT   14 _fini
```

可以看到，有 `main()` 函数，`probably_inlined()` 函数，没有了 `unused()` 函数。因为虽然 `unused()` 函数在 tu2.c 中定义了，但是实际上并没有它并没有被调用，所以该函数是个死函数，所以在 LTO 时会被删除。

我们可以再看一下，不开启 LTO 时编译 tu1.c 和 tu2.c 得到可执行文件 main.nonlto：

```
% clang -fuse-ld=lld tu1.c tu2.c -o main.nonlto
% readelf -sW ./main.nonlto | awk '$4 == "FUNC"'
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND
__libc_start_main@GLIBC_2.2.5 (2)
     3: 00000000002015f0     0 FUNC    LOCAL  DEFAULT   12
deregister_tm_clones
     4: 0000000000201620     0 FUNC    LOCAL  DEFAULT   12
register_tm_clones
     5: 0000000000201660     0 FUNC    LOCAL  DEFAULT   12
__do_global_dtors_aux
     8: 0000000000201690     0 FUNC    LOCAL  DEFAULT   12 frame_dummy
    16: 00000000002015e0     2 FUNC    LOCAL  HIDDEN    12
_dl_relocate_static_pie
    21: 0000000000201740     2 FUNC    GLOBAL DEFAULT   12 __libc_csu_fini
    22: 00000000002015b0    43 FUNC    GLOBAL DEFAULT   12 _start
    23: 00000000002016d0   101 FUNC    GLOBAL DEFAULT   12 __libc_csu_init
    24: 00000000002016a0     5 FUNC    GLOBAL DEFAULT   12 main
    27: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND
__libc_start_main
    30: 0000000000201744     0 FUNC    GLOBAL DEFAULT   13 _init
    31: 000000000020175c     0 FUNC    GLOBAL DEFAULT   14 _fini
    34: 00000000002016c0     4 FUNC    GLOBAL DEFAULT   12 probably_inlined
    35: 00000000002016b0     4 FUNC    GLOBAL DEFAULT   12 unused
```

可以看到 `unused()` 函数被保留在了最终的可执行文件中。

通过这个例子，我们看到了 LTO 可以在链接时 remove dead symbol。

实际上，如果我们还可以通过 optimization remarks 得到在 LTO 优化时都删除了哪些函数：

```
% clang -flto -fuse-ld=lld -Wl,--opt-remarks-passes,lto -Wl,--opt-remarks-
filename,main.lto.yaml tu1.c tu2.c -o main
```

这里我们只保留了与 lto 相关的 optimization remarks，默认生成的 optimization remarks 是 YAML 格式文件，该文件 main.lto.yaml 的内容如下：

```
--- !Passed
Pass:            lto
Name:            deadfunction
```

```
Function:        unused
Args:
  - Function:        unused
  - String:          ' not added to the combined module '
...
```

从 main.lto.yam 文件的内容也可以看出来 `unused()` 函数在 lto 优化阶段被删除掉了。

Inside the source code

下面我们了解一下 LTO remove dead symbol 的代码实现。

这里给出使用 lld 作为 linker，链接过程执行到 remove dead symbol 所经过的函数：

```
=> void LinkerDriver::linkerMain(ArrayRef<const char *> argsArr) at
lld\ELF\Driver.cpp:475
===> void LinkerDriver::link(opt::InputArgList &args) at
lld\ELF\Driver.cpp:2165
=====> void LinkerDriver::compileBitcodeFiles() at lld\ELF\Driver.cpp:1979
=======> std::vector<InputFile *> BitcodeCompiler::compile() at
lld\ELF\LTO.cpp:299
=========> Error LTO::run(AddStreamFn AddStream, NativeObjectCache Cache)
at llvm\lib\LTO\LTO.cpp:995
===========> void llvm::computeDeadSymbolsWithConstProp(...) at
llvm\lib\Transforms\IPO\FunctionImport.cpp:956
===========> Error LTO::runRegularLTO(AddStreamFn AddStream) at
llvm\lib\LTO\LTO.cpp:1043
=============> Error LTO::linkRegularLTO(RegularLTOState::AddedModule Mod,
bool LivenessFromIndex) at llvm\lib\LTO\LTO.cpp:853
```

根据函数名也可以看出，计算 dead symbol 的核心函数就是 `void llvm::computeDeadSymbolsWithConstProp(...)` ，实现如下：

```
llvm-project\llvm\lib\Transforms\IPO\FunctionImport.cpp:955
955: // Compute dead symbols and propagate constants in combined index.
956: void llvm::computeDeadSymbolsWithConstProp(
957:     ModuleSummaryIndex &Index,
958:     const DenseSet<GlobalValue::GUID> &GUIDPreservedSymbols,
959:     function_ref<PrevailingType(GlobalValue::GUID)> isPrevailing,
960:     bool ImportEnabled) {
961:   computeDeadSymbols(Index, GUIDPreservedSymbols, isPrevailing);
962:   if (ImportEnabled)
963:     Index.propagateAttributes(GUIDPreservedSymbols);
964: }
```

函数 `computeDeadSymbols()` 的实现如下：

核心算法就是不动点的计算：将 GUIDPreservedSymbols 对应的 retained symbol 标记为 live，作为 worklist 的初始值。然后不断遍历 worklist 中的每一个 symbol，将该 symbol 引用的其他 symbol 标记为 live 的，加入到 worklist 中。一直迭代，直至没有新的被标记为 live 的 symbol。

在函数 `computeDeadSymbols()` 实现该 worklist 算法时，是用类似栈的方式处理的：将新标记为 live 的 symbol 入栈，然后不断处理栈顶的 symbol，该栈顶 symbol 出栈，将该 symbol 引用的其他之前没有添加过 worklist 中的 symbol 标记为 live 的，加入到栈顶。一直迭代，直至栈为空。

```
llvm-project\llvm\lib\Transforms\IPO\FunctionImport.cpp:842
842: void llvm::computeDeadSymbols(
843:     ModuleSummaryIndex &Index,
844:     const DenseSet<GlobalValue::GUID> &GUIDPreservedSymbols,
845:     function_ref<PrevailingType(GlobalValue::GUID)> isPrevailing) {
846:   assert(!Index.withGlobalValueDeadStripping());
847:   if (!ComputeDead)
848:     return;
849:   if (GUIDPreservedSymbols.empty())
850:     // Don't do anything when nothing is live, this is friendly with tests.
851:     return;
852:   unsigned LiveSymbols = 0;
853:   SmallVector<ValueInfo, 128> Worklist;

       第 854 - 873 行初始化 worklist

854:   Worklist.reserve(GUIDPreservedSymbols.size() * 2);
855:   for (auto GUID : GUIDPreservedSymbols) {
856:     ValueInfo VI = Index.getValueInfo(GUID);
857:     if (!VI)
858:       continue;
859:     for (auto &S : VI.getSummaryList())
860:       S->setLive(true);
861:   }
862:
863:   // Add values flagged in the index as live roots to the worklist.
864:   for (const auto &Entry : Index) {
865:     auto VI = Index.getValueInfo(Entry);
866:     for (auto &S : Entry.second.SummaryList)
867:       if (S->isLive()) {
868:         LLVM_DEBUG(dbgs() << "Live root: " << VI << "\n");
869:         Worklist.push_back(VI);
870:         ++LiveSymbols;
871:         break;
872:       }
873:   }
```

874:

visit 判断当前处理的 symbol 是否在已经被标记为 live，即之前已经加过 worklist 中被处理过了。

如果没有，则将其标记为 live，然后添加到 worklist 中。

```
875:   // Make value live and add it to the worklist if it was not live before.
876:   auto visit = [&](ValueInfo VI, bool IsAliasee) {
877:     // FIXME: If we knew which edges were created for indirect call profiles,
878:     // we could skip them here. Any that are live should be reached via
879:     // other edges, e.g. reference edges. Otherwise, using a profile collected
880:     // on a slightly different binary might provoke preserving, importing
881:     // and ultimately promoting calls to functions not linked into this
882:     // binary, which increases the binary size unnecessarily. Note that
883:     // if this code changes, the importer needs to change so that edges
884:     // to functions marked dead are skipped.
885:     VI = updateValueInfoForIndirectCalls(Index, VI);
886:     if (!VI)
887:       return;
888:
889:     if (llvm::any_of(VI.getSummaryList(),
890:                       [](const std::unique_ptr<llvm::GlobalValueSummary> &S) {
891:                         return S->isLive();
892:                       }))
893:       return;
894:
895:     // We only keep live symbols that are known to be non-prevailing if any are
896:     // available_externally, linkonceodr, weakodr. Those symbols are discarded
897:     // later in the EliminateAvailableExternally pass and setting them to
898:     // not-live could break downstreams users of liveness information (PR36483)
899:     // or limit optimization opportunities.
900:     if (isPrevailing(VI.getGUID()) == PrevailingType::No) {
901:       bool KeepAliveLinkage = false;
902:       bool Interposable = false;
903:       for (auto &S : VI.getSummaryList()) {
904:         if (S->linkage() == GlobalValue::AvailableExternallyLinkage ||
905:             S->linkage() == GlobalValue::WeakODRLinkage ||
906:             S->linkage() == GlobalValue::LinkOnceODRLinkage)
907:           KeepAliveLinkage = true;
908:         else if (GlobalValue::isInterposableLinkage(S->linkage()))
909:           Interposable = true;
910:       }
911:
912:       if (!IsAliasee) {
913:         if (!KeepAliveLinkage)
914:           return;
915:
916:         if (Interposable)
917:           report_fatal_error(
918:               "Interposable and available_externally/linkonce_odr/weak_odr "
919:               "symbol");
920:       }
921:     }
922:
923:     for (auto &S : VI.getSummaryList())
924:       S->setLive(true);
925:     ++LiveSymbols;
926:     Worklist.push_back(VI);
927:   };
928:
```

迭代直至 worklist 为空，即没有新的 symboal 被标记为 live，添加至 worklist 中

```
929:   while (!Worklist.empty()) {
930:     auto VI = Worklist.pop_back_val();
931:     for (auto &Summary : VI.getSummaryList()) {
932:       if (auto *AS = dyn_cast<AliasSummary>(Summary.get())) {
933:         // If this is an alias, visit the aliasee VI to ensure that all copies
934:         // are marked live and it is added to the worklist for further
935:         // processing of its references.
936:         visit(AS->getAliaseeVI(), true);
937:         continue;
938:       }
939:       for (auto Ref : Summary->refs())
```

LLVM-Clang-Study-Notes / source / lto / RemoveDeadSymbol.rst ↑ Top

Preview Code Blame Raw

```
944:     }
945:   }
946:   Index.setWithGlobalValueDeadStripping();
947:
948:   unsigned DeadSymbols = Index.size() - LiveSymbols;
949:   LLVM_DEBUG(dbgs() << LiveSymbols << " symbols Live, and " <<
DeadSymbols
950:                      << " symbols Dead \n");
951:    NumDeadSymbols += DeadSymbols;
952:    NumLiveSymbols += LiveSymbols;
953: }
```

这里再次用在 Example 节中的例子来分析该函数 `computeDeadSymbols()` ：

1. 第 854 - 873 行初始化 Worklist，对于 Example 节中的例子来说，Worklist 中此时只有一个元素，就是 `main()` 函数对应的 ValueInfo

```
(gdb)
864       for (const auto &Entry : Index) {
(gdb)
927       };
(gdb) p Worklist.size()
$28 = 1
(gdb) p Worklist.begin()->name().str()
$29 = "main"
```

2. 第 929 - 945 行第一轮迭代：因为 `main()` 函数调用了 `probably_inlined()` 函数，所以会执行第 943 行： `visit(Call.first, false);` 此时 Call.first 就是 `probably_inlined()` 函数对应的 ValueInfo

```
929       while (!Worklist.empty()) {
(gdb)
930         auto VI = Worklist.pop_back_val();
(gdb)
931         for (auto &Summary : VI.getSummaryList()) {
(gdb)
932           if (auto *AS = dyn_cast<AliasSummary>(Summary.get())) {
(gdb)
939           for (auto Ref : Summary->refs())
(gdb)
941           if (auto *FS = dyn_cast<FunctionSummary>(Summary.get()))
(gdb)
942             for (auto Call : FS->calls())
(gdb)
943               visit(Call.first, false);
(gdb) p Call.first.name().str()
$31 = "probably_inlined"
```

3. 第 876 - 927 行处理 `probably_inlined()` 函数对应的 ValueInfo ，因为 `probably_inlined()` 函数对应的 ValueInfo 不是 live 的，没有添加进 Worklist 中过，所以在将其设置为 live，然后添加至 Worklist 中

```
876       auto visit = [&](ValueInfo VI, bool IsAliasee) {
(gdb) n
885         VI = updateValueInfoForIndirectCalls(Index, VI);
(gdb)
886         if (!VI)
(gdb)
889         if (llvm::any_of(VI.getSummaryList(),
(gdb)
900         if (isPrevailing(VI.getGUID()) == PrevailingType::No) {
(gdb)
923         for (auto &S : VI.getSummaryList())
(gdb)
924           S->setLive(true);
(gdb)
923         for (auto &S : VI.getSummaryList())
(gdb)
925         ++LiveSymbols;
(gdb)
926         Worklist.push_back(VI);
(gdb)
927       };
```

4. 第 929 - 945 行第二轮迭代，此时 Worklist中还是只有一个元素，是 `probably_inlined()` 函数对应的 ValueInfo，而 `probably_inlined()` 函数没有引用其他的 symbol，所以在没有添加任何 symbol 至 Worklist 中。第 929 - 945 行第三轮迭代，Worklist 为空，到达不动点，迭代结束。

```
929       while (!Worklist.empty()) {
(gdb) n
930         auto VI = Worklist.pop_back_val();
(gdb)
931         for (auto &Summary : VI.getSummaryList()) {
(gdb)
932           if (auto *AS = dyn_cast<AliasSummary>(Summary.get())) {
(gdb)
939           for (auto Ref : Summary->refs())
(gdb)
941           if (auto *FS = dyn_cast<FunctionSummary>(Summary.get()))
(gdb)
942             for (auto Call : FS->calls())
(gdb)
931         for (auto &Summary : VI.getSummaryList()) {
```

```
(gdb)
929       while (!Worklist.empty()) {
(gdb)
946       Index.setWithGlobalValueDeadStripping();
```

5. 函数 `computeDeadSymbols()` 结束，tu1 和 tu2 中一共有 3 个 symbol，其中 `main()` 和 `probably_inlined()` 是 live 的，而 `unused()` 是 dead，所以最后链接时，会删除 `unused()` 函数。

```
946       Index.setWithGlobalValueDeadStripping();
(gdb)
948       unsigned DeadSymbols = Index.size() - LiveSymbols;
(gdb)
949       LLVM_DEBUG(dbgs() << LiveSymbols << " symbols Live, and " <<
DeadSymbols
(gdb)
951       NumDeadSymbols += DeadSymbols;
(gdb)
952       NumLiveSymbols += LiveSymbols;
(gdb)
853       SmallVector<ValueInfo, 128> Worklist;
(gdb)
953     }
(gdb) p DeadSymbols
$32 = 1
(gdb) p LiveSymbols
$33 = 2
```

References

1. https://en.wikipedia.org/wiki/Interprocedural_optimization
2. http://llvm.org/docs/LinkTimeOptimization.html
3. http://llvm.org/docs/GoldPlugin.html