ThinLTO Symbol Linkage and Renaming

1 Introduction

This document describes handling of symbols that may need linkage type changes or renaming to support ThinLTO importing.This applies to both the symbol in its original module as well as in the module importing it.

In LLVM, the GlobalValue class is used to represent function, variable and alias symbols. The Function and GlobalVariable classes are both derived from the GlobalObject class which itself is derived from GlobalValue. The GlobalAlias class is derived directly from GlobalValue. Note that LLVM GlobalValues include static variables and functions in C/C++.

During ThinLTO importing of a function from another module, the symbols from that module are parsed and imported as either a declaration or definition. Note that while we are importing one function at a time from another module, we typically always import all variable definitions (the one exceptions are variables with AppendingLinkage and WeakAnyLinkage, as described in Section 2.2).

The following sections discuss the handling of values with the given original linkage types (in the original module). The effects on the linkage type in both the original module and in the imported copy in another module are described.

2 Non-Discardable Values

Non-discardable values are those that cannot be discarded by a module even when the value is unreferenced, i.e. it may be referenced by another module. This includes all linkage types except local (internal or private) and linkonce. For these values (both variables and functions), a reference may be imported into another module without fear that its definition may be eliminated in the original module. As a result, when the definitions of these non-discardable values are imported, the imported copy may safely be eliminated after optimizations such as inlining, as a copy is guaranteed to be available in the original module. In practice we may not eliminate these imported definitions, as discussed below, depending on the linkage type handling in the imported copy.

2.2 Linkage Effects

There is no change to the linkage type required in the original module for any non-discardable values that may be imported to another module.

During importing, ideally all non-discardable definitions have their linkage type changed in the imported copy to AvailableExternallyLinkage. This signals to the compiler that the definition can safely be eliminated after inlining (i.e. by the new EliminateAvailableExternally pass). In practice, we do not change the linkage for all imported non-discardable defs. The linkage effects for the non-discardable linkage types are described below:

2.2.2 ExternalLinkage

An imported ExternalLinkage definition can be changed to AvailableExternallyLinkage. If it is eliminated later by the EliminateAvailableExternally pass, the resulting decl becomes ExternalLinkage.

2.2.1 WeakAnyLinkage

Importing a WeakAnyLinkage definition could change the result of the program as it could cause a different weak definition to be selected by the linker. WeakAnyLinkage can be specified via __attribute__ ((weak)) on a function definition, to allow overriding by a strong definition in another module. If no strong definition exists, the linker will select the first weak definition. Importing a weak definition into a different module can change the order the weak defs are seen

by the linker and change the program semantics. Therefore, any WeakAnyLinkage definitions are only imported as declarations, which are given ExternalWeakLinkage. WeakAny aliases are handled similarly (imported as ExternalWeakLinkage aliases).

2.2.2 WeakODRLinkage

For WeakODRLinkage, there is a guarantee that all copies will be equivalent, so the issue described above for WeakAny does not exist, and the definition can be imported. For WeakODRLinkage, the imported definition should retain the original WeakODRLinkage. If imported as a declaration, it should instead have ExternalWeakLinkage.

WeakODRLinkage symbols cannot be marked AvailableExternallyLinkage, because if the def is later dropped (by the EliminateAvailableExternally pass), the new decl is marked ExternalLinkage. For these weak symbols, however, the correct linkage for the decl is actually ExternalWeakLinkage, so that they get treated appropriately by the linker. But the information about their original weak linkage would be gone once they were changed to AvailableExternallyLinkage. For now, since weak symbols are expected to be uncommon, we will leave these symbols with their original weak linkage, which means they will not be discardable in the imported destination. If this becomes a problem, we can investigate retaining information about the original linkage type.

2.2.3 AppendingLinkage

This applies to special variables such as the global constructors and destructors lists. We never import these as they would get executed multiple times, which is incorrect.

2.2.4 CommonLinkage

Since common symbols are always zero-initialized variables, they do not take up room. It is simplest to import these defs as common.

3 Linkonce Values

The LinkOnceODRLinkage and LinkOnceAnyLinkage types refer to linkonce linkage, which allows merging of different globals with the same name. Unreferenced linkonce globals may also be discarded. Linkonce values include some COMDAT functions (COMDAT may also have Weak linkage) and vtable variables. For linkonce values, duplicates are allowed and the linker selects one.

3.1 Linkage Effects

Since duplicates are allowed and the linkonce values are already discardable, imported linkonce values can remain linkonce in the imported copy. Any duplicate imported copy will be handled by the linker, and it may remain discardable in the importing module if it isn't referenced after importing/inlining.

Similarly, there is also no change in linkage type required in the original module.

3.2 Importing Strategy

The main issue with linkonce values is that they are discardable in the original module (e.g. if all references are inlined in the original module). However, ThinLTO importing may introduce a cross-module reference to a linkonce value in the original module. Care must be taken to ensure that such a reference imported into another module is satisfied at link time by a definition somewhere. To handle this for linkonce functions, the ThinLTO importer must force-import any linkonce functions referenced by another imported function. To do this, after importing a function, the ThinLTO importer walks all newly imported operations looking for references to functions with linkonce linkage type. Any found are also imported, along with functions and variables in the same COMDAT group (note the COMDAT group must always be imported in its entirety regardless of whether it has linkonce or weak linkage). For linkonce variables, since GlobalVariable definitions are always imported when we import a function from the same module as described in the introduction, the linkonce variable definitions are therefore imported and available.

4 Static Values

File static functions and variables have local linkage types (i.e. internal or private), and are discardable. These values need special handling during importing as described below. The local linkage types InternalLinkage and PrivateLinkage (private is the same as internal but is omitted from the symbol table) are handled the same for ThinLTO.

4.1 Static Variables

Read-write or address taken static variables must always be promoted to global scope (i.e. non-discardable, non-local linkage) if they are imported to another module. This is important to

ensure that one single copy of the static variable is used in address comparisons or when updating its value.

The mechanics of static promotion are described below in Section 4.3.

A static (local linkage) variable marked as a read-only (marked constant by LLVM) need not be promoted/renamed, unless it is address taken as described above. The imported copy of its definition can simply be used by any imported references in the read-only non-address taken case.

4.2 Static Functions

When a reference to a static (local linkage) function is imported, in order to satisfy the imported reference, there are two possible strategies:

1. The function must be promoted to global scope (i.e. non-local linkage) both in its original module (where it is no longer discardable) and in the importing module; OR
2. The static function definition must also be imported into the same importing module, so that a local copy is available for the reference.

The strategy to be used for a particular function will ultimately depend on several factors such as whether it is address taken, its size, etc. This is discussed in more detail further on.

Note that address taken static functions must always be promoted for correctness. The reason can be seen with the following example:

```
a.cc:
   static void foo() { … };
   static funcptr P;

   void bar() {
         if (P == &foo) {
            ...
         }
   }

   void baz() {
         P = &foo;
   }

b.cc:
  … bar(); …
```

If bar() is imported into b.cc (and gets inlined), then b.cc will have a reference to static foo(). It we then import foo() and leave the imported copy local/unpromoted, the imported P==&foo comparison will fail (assume baz() is not referenced by or imported into b.cc). That is because P will point to the a.cc::foo() and the imported comparison will compare against the imported copy b.cc::foo(). This is avoided by promoting address taken static functions, in both their original module (if the function is in the global function index) and when it has been imported along with an address taken reference into another module. Similar logic applies to read-only static variables as mentioned in Section 4.1.

The mechanics of static promotion are described later in Section 4.3.

It is important to note that the promotion decisions between the original module and the importing module must be consistent (it is ok for the function to be promoted in the original module but not in the importing module, but not vice versa, or we will have an undefined reference reference at link time). However, the original module and importing modules are compiled independently through the ThinLTO phase-3 parallel backend. There are several ways to ensure consistent promotion. Two of the simplest are:

1. Always promote imported static functions
    a. When importing a static function reference or definition, always promote the imported value
    b. In the original module, promote any static functions that may be imported into another module (any static functions in the global function index/summary map)
2. Minimize static promotions by only promoting address taken static functions
    a. When a function is imported, look through newly imported instructions and force import any referenced static functions.
    b. When importing is complete, any imported static functions that are address taken must be promoted
    c. In the original module, promote any static functions that are address taken.

Strategy 1 (always promote) has the disadvantage in that it potentially results in a larger symbol table, due to promoted functions no longer being discardable. However, in practice (based on cpu2006 measurements) this does not turn out to be a significant issue.

Strategy 2 (promote only address taken) has the disadvantage in that it can increase code size, in the case where imported unpromoted static functions are not inlined into all call sites in the imported module, and we end up with multiple copies. This turns out to be a more significant issue for a couple of cpu2006 benchmarks, notably 400.perlbench. However, this can be alleviated by enabling function sections and linker garbage collection.

A third intermediate strategy would be to promote static functions that are either address-taken or unlikely to be fully inlined. Large or cold static functions are unlikely to be inlined, and are therefore unlikely to be imported anyway. Therefore, the plugin should not even include these functions in the combined function summary (based on metrics such as function size or

hotness). Any static function that is not in the combined index will therefore not be eligible for importing, and should be promoted. So both the original module and the importing module will consult the combined function summary for each static function to see whether it should be static promoted (if entry not found) vs force imported (if entry found). This can be refined further to static promote a address taken and/or non-importable static function in the original module only if it is referenced by a function that is found in the map (because the reference may be imported into another module). This is summarized below:

3. Promote only address taken static functions and those not enabled for importing by the plugin
    a. When a function is imported, look through newly imported instructions and identify any referenced static functions. If the referenced static function is found in the function summary, force import it. Otherwise, promote it.
    b. When importing is complete, any imported (and therefore unpromoted) static functions that are address taken must be promoted
    c. In the original module, promote any static functions that either:
        i. are address taken; or
        ii. may not be imported into another module (any static functions not in the global function index/summary map)

        iff they are referenced by a function that may be imported into another module.

Given the results of experiments with cpu2006, we will initially implement strategy 1 (always promote). This is also the simplest strategy.

4.3 Static Promotion

Promoting a static variable or function to non-local/non-static requires two main changes: changing the linkage and ensuring consistent naming between the definition in the original module and references in importing modules.

4.3.1 Linkage Effects

The linkage of static functions and variables are changed are as follows:

- Definition in the original module: ExternalLinkage
- Definition in the importing module: AvailableExternallyLinkage
- Declaration in the importing module (i.e. def not imported): ExternalLinkage

Note that in the importing module the new linkage type of the promoted function/variable definition is the same as for non-discardable (non-local, non-linkonce) ExternalLinkage symbols as described in Section 2.

4.3.2 Renaming

Since there may be multiple static functions from different modules with the same name before importing/promotion, or a global function which already has the same name, the promoted static functions must be renamed to avoid naming conflicts. It is important that the promoted definition in the original module is given the same name as the promoted reference in the importing module, so that the reference in the importing module can be satisfied by the original module at link time.

To do the renaming consistently, we can include a module-specific identifier in the new name. The plugin step in phase-2 of ThinLTO (which builds the combined function index/summary) has visibility into all modules included in the ThinLTO build. It can simply number the modules and record the assigned module ID in the function summary information (either along with each function from that module, or along with the module name which will be shared in a module string table for efficiency). Then during promotion, the module ID in the combined function index/summary can be consulted and appended to the original name, along with an LLVM-specific suffix identifying this as a promoted static.

For example,

```
        static void foo();  // In module with ID 1
```

originally has bitcode definition in module ID 1:

```
        define internal void @foo() // InternalLinkage
```

which becomes after promotion/renaming:

```
        define void @foo.llvm.1()  // ExternalLinkage, new suffix ".llvm.1"
```

When we import a static definition into another module (say module ID 2), before promotion it has declaration:

```
        define internal void @foo()     // InternalLinkage, func summary info: from Module ID 1
```

which becomes after promotion/renaming:

```
        define available_externally void @foo.llvm.1()  // AvailableExternallyLinkage, new suffix ".llvm.1"
```

If a reference is imported but not the definition, before promotion the new declaration is:

```
        declare internal void @foo() // InternalLinkage, func summary info: from Module ID 1
```

which becomes after promotion/renaming:

```
        declare void @foo.llvm.1()  // ExternalLinkage, new suffix ".llvm.1"
```

Note that the new EliminateAvailableExternallyPass (under review) will change the linkage type from AvailableExternallyLinkage to ExternalLinkage on the declaration it leaves for any eliminated available externally definitions, which is consistent with the above behavior.

5 Linkage Change Summary Table

The following table summarizes the linkage change that happen during ThinLTO backend compilations. Note that the only time the linkage changes in the original module is for the static promotion case (for Internal and Private linkages), where it changes to External as described in Section 4.3.1. In all other cases the linkage in the original module stays the same, and therefore is not noted in the table below.

Note that there are no InternalLinkage, PrivateLinkage, AvailableExternallyLinkage, LinkOnce*Linkage, Weak*Linkage, AppendingLinkage and CommonLinkage declarations. There are no ExternalWeakLinkage definitions.

Original Module		Importing Module Linkage	
		Import definition	Import as declaration
External (def)		AvailableExternally	External
External (decl)		N/A	External
Internal (def)	Promote[1]	AvailableExternally	External
	NoPromote	Internal	N/A (force import def)
Private (def)	Promote[2]	AvailableExternally	External
	NoPromote	Private	N/A (force import def)
AvailableExternally (def)		AvailableExternally	External
LinkOnceAny (def)		LinkOnceAny	N/A (force import def)
LinkOnceODR (def)		LinkOnceODR	N/A (force import def)
WeakAny (def)		- (never import)	ExternalWeak
WeakODR (def)		WeakODR	ExternalWeak
Appending (def) (only variables)		N/A (never import)	N/A (never import)
ExternalWeak (decl)		N/A	ExternalWeak
Common (def) (only variables)		Common	N/A (always import def)

[1] Linkage in original module changes to External
[2] Linkage in original module changes to External