Rationale for Link Time Optimization in GCC

Kenneth Zadeck
NaturalBridge, Inc.
zadeck@naturalbridge.com

Roadmap

- What is Link Time Optimization (LTO)?
- Why do LTO?
- Who will use LTO?
- Why does LTO work.
- LTO Requirements.
- Making LTO work.
- Making LTO work well.
- Who is working on LTO.

What is Link Time Optimization (LTO)?

- Conventional Compilation:
  (1) Compile and optimize one file at a time.
  (2) Link the results
- LTO Compilation
  (1) Convert source language to IL one file at a time.
  (2) Link the IL together, then compile and optimize and link everything together.

Why Do LTO?

- LTO will provide more performance for certain applications.
  - On average we can expect 10-15%.
  - There is a lot of variation.
    - Function calls inside of inner loops.
    - Heavily abstracted C++ will show the most improvement.
    - Hand optimized Fortran will show little improvement.
    - More improvement on in-order machines.
- Everyone else is doing LTO.
  - GCC must remain a viable compiler.

Who Will Use LTO?

- The UNIX distributions:
  - Any large, complex system is a good candidate.
  - Systems that have made extensive use of data abstraction.
- A large user community will require a good implementation:
  - Easy to use.
  - Relatively bug free.
  - Good compile time.
- The benchmarking/research community.

Why Does LTO Work?

Module a

```
static b,c;
int foo (…) {
   while (aLongTime) {
      … = b + c
      … = bar ()
      … = b + c
   }
}
void baz () {
 b = 6;
}
```

Module b

```
int bar () {
```

what can be here?

```
}
```

If there is a direct or indirect call to `baz` then `b+c` cannot be commoned.

Why Does LTO Work?

- Gets rid of function calls.
  - Try to make it so that inner loops are call free.
  - Get rid of trivial access functions.
- Allows wider context for analysis.
  - Better alias analysis.
  - Generally not the whole world.
- Better scheduling.
  - Important for in order machines.
  - Not so for out of order machines.

Requirements

- Should require very small changes to the program's tool-chain.
  - Users do not like new tools.
- Allow mixing files from different languages.
- Retain *linker semantics*.
- Deal with different compile time options.

Very Small Changes to the Toolchain

- The minimum conversion to use LTO:
  - Only one option added to compile command.
  - Only one option added to link command.
  - No additional files created.
  - No additional files required.
- More options will be available for the advanced user.
  - Many errors can be detected when you see how the program fits together.

Allow Mixing Files from Different Languages

- Given that everything is gimple, we should just be able to inline one languages gimple into another.
- In practice, this is very hard and will require a lot of work.
- Most languages do not define how they work with other languages.
  - There are known ways of doing this: most of them bad.

Retain Linker Semantics

- If the program worked without LTO, it should work with LTO.
- Most programming languages do not define that all external variables be defined exactly the same.
  - If `x` is declared as a `long` in one module and an `int` in another, how should it work?
- With additional options we can increase the quality of warning messages.

Deal with Different Compile-time Options

- Some combinations are just not possible:
  - Options that control which passes are run will most likely be ignored.
  - Options that control which exception models are used may be impossible.
- GCC currently has too many options.

Making LTO Work

Conceptually easy:

- Pickle out the intermediate language.
- Gather the intermediate language files together.
- Run the existing optimizations over the combined program.
- Add new optimizations.
- Produce one big executable.
- Celebrate the success.

In practice there are many hard problems.

- Some are research, some are engineering.

Pickling Out the Intermediate Code – Types and Global Variables

- We have chosen to extend DWARF-3.
  - This has the added benefit of enhancing the debugging information.
- This might have been a mistake.
  - Dwarf-3 is bulky, especially if extended.
  - It is not random access.

Pickling Out the Intermediate Code – Statements

- We use a custom format to encode the statements.
  - This is compact.
  - Allows random access to function bodies.
- Operators trivially encode current gimple operations.
  - Some front ends still generate a little rtl.

Pickling Out the Intermediate Code – Output Will Go into Current .o Files

- This has the smallest tool-chain impact.
- Can make for large .o files.
- Since we are using the .o files, we can steal linker technology to find all of the .o files.
- These are then passed to new phase of the compiler to compile and then that will go the linker.
- Not much rocket science needed here.

Running Existing Optimization Passes

- Many existing optimizations will work with LTO
  - Some experience has been gained by the existing --*combine* mode.
- Space problems abound:
  - GCC is a pig.
  - We should not expect to be able to hold the entire program in memory while we are compiling.
- Some optimizations may not scale properly.

Making LTO Work Well

- Gimple does not encode all of the semantics of a program.
- GCC has a poor/nonexistent internal type system.
- Space.
- Many passes will not scale for entire programs.
- We will need some sort of parallel compilation.

Gimple Does Not Encode all of the Semantics of a Program

- GCC relies on lang-hooks avoid representing many parts of the semantics.
- There is no encoding in gimple for many compiler options.
- This poses problems when we begin to inline from one compilation unit to another.
- Everything must be encoded into the gimple semantics.
  - This is a lot of work.

Types I – Incompatible Modules

- Are the two types the same?
- It is not uncommon for a type in file A to be declared differently from a type with the same name in file B.
- What if

```
sizeof(int) != sizeof (long)?
```

- The linker can many times make this work.
- We need to have the same semantics, at least by default.
- We may need to inhibit inlining where the one function has one view of a variable and the other function has another view of the variable.

Types II – The Front Ends Make All the Decisions

- The types are not encoded inline.
- GCC falls back on the front ends to say if two types are compatible.
  - Requires removal of all lang-hooks.
  - Need language independent symbol table.

Types III – Inter-language Type Equivalence

- No language spec talks about how it's types map onto other languages types.
  - Can a class match a struct?
  - What if they are not layout compatible?
  - Can there be two definitions for the same function?
  - Which should be called?

Types – Possible Solutions

- The obvious solution is to use structural equivalence with the types represented as graphs of low-level primitives.
- This allows the question to generally be answered but at a cost:
  - Anything that looks the same is considered to be the same. This degrades the information produced by alias/type inheritance analysis.
- This is an hard open research problem.

Space

- It is not the final frontier, but it is a significant problem.
- We currently have people working on many of the space issues.
- We are not going be able to compile large systems simply by cleaning things up.
  - But the system will be usable until we get a real parallel compilation system.
- This is something that is overdue for GCC.

Many Passes Will Not Scale Properly

- Phases must separate analysis from transformation.
  - Call graph analysis currently requires the function bodies as well is the call graph.
    - This will have to be uncoupled.
- In general, no interprocedural analysis pass can assume access to the function bodies.
  - We will have to produce *compact* summaries of each function that are written into the .o files.
    - Hopefully, different phases can share summaries.

We Will Need Parallel Compilation

- It is not reasonable to expect that the entire program will be compiled on one processor.
- Must take advantage of some parallelism or compilation will be too expensive.
  - There is a vast amount of parallelism possible by letting a network of machines compile individual functions.
  - This needs to be carefully designed so that we do not swamp networks or file systems.
  - The design cannot expect each farm machine to have access to every .o file in the compilation.

Status

- Reading and writing gimple.
  - Basically finished.
- Reading and writing types.
  - Close, but still work being done.
- Langhooks
  - Many have been converted.
  - Need a master plan for types.

Who is working on LTO

- The encoder – decoder for LTO
  - Mark Mitchell, Jim Bandy, Nathan Froyd @ CodeSourcery
  - Kenneth Zadeck @ NaturalBridge
- The LTO Front End
  - Bill Maddox @ Google
- Tuples Conversion
  - Diego Novillo, Chris Matthews @ Google,
  - Aldy Hernandez @ Redhat
- Lang Hook Removal
  - Ollie Wild, Rafael Espindola, Robert Keenedy @ Google
- Types
  - Olga Golovanevsky @ IBM, Richard Gunther @ SUSE