ThinLTO

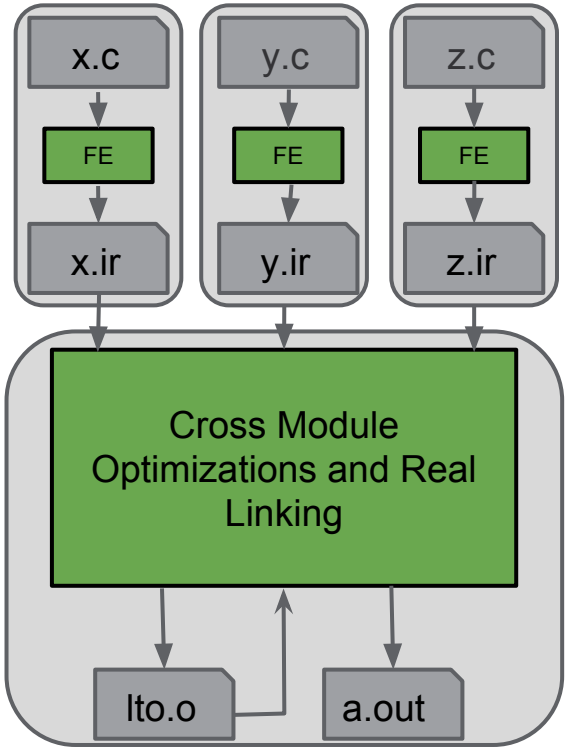
A Fine-Grained Demand-Driven Infrastructure

Teresa Johnson, Xinliang David Li
tejohnson,davidxl@google.com

Outline

- CMO Background
- ThinLTO Motivation and Overview
- ThinLTO Details
- Build System Integration
- LLVM Prototype Status
- Preliminary Experimental Results
- Next Steps

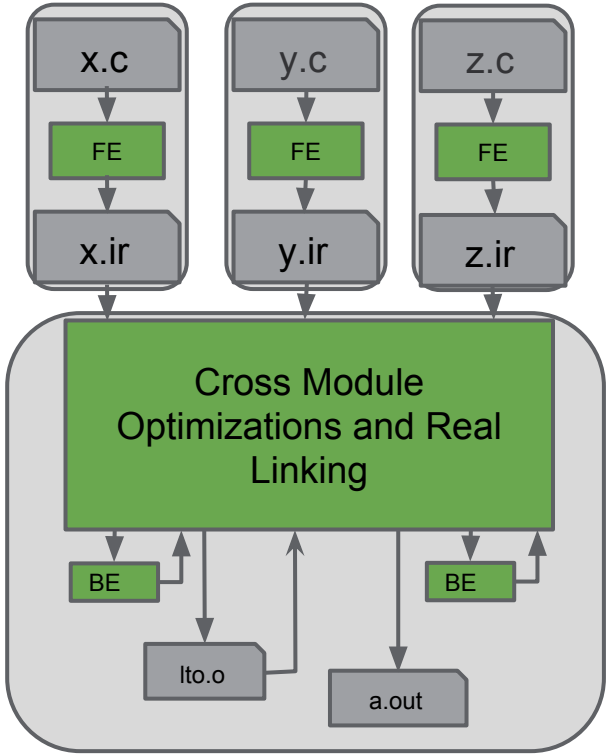
Cross-Module Optimization (CMO) Background



Monolithic LTO (Link-Time Optimization):

- Frontend parsing and IR generations are all done in parallel
- Cross Module Optimization is done at link time via a linker plugin
- CMO pass consumes lots of memory and is generally not parallel
- CMO is done in linker process
- Doesn't scale to very large applications

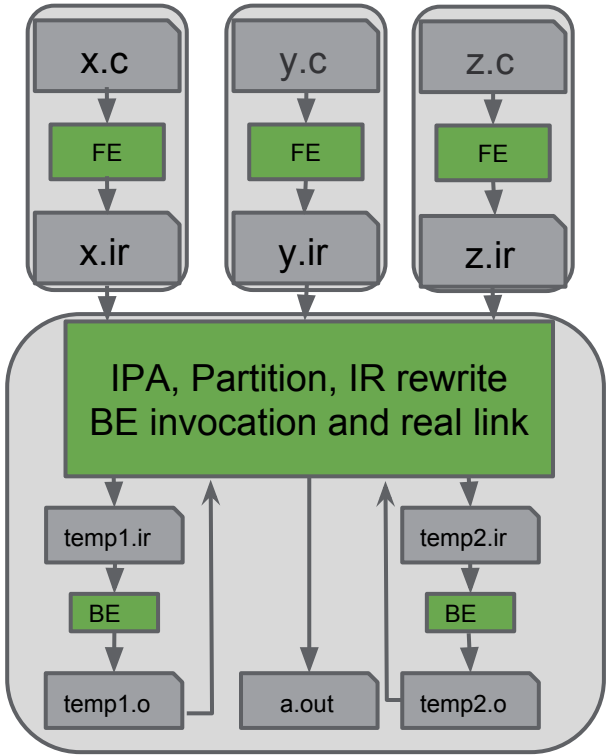
CMO Background (cont)



Monolithic LTO with Parallel BE:

- CMO still performed in serial
- Intramodule optimization and code generation done in parallel (thread or process level)
- Example: HPUX Itanium compiler (SYZYGY framework)

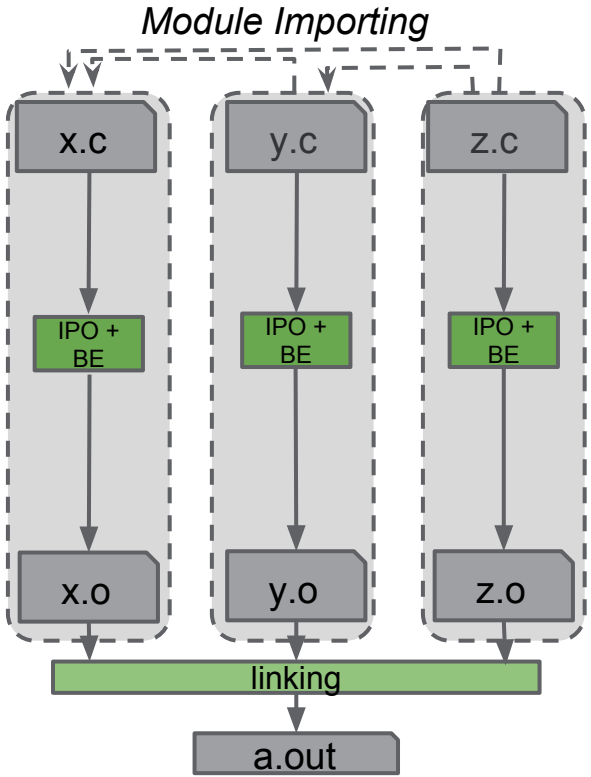
CMO Background (cont)



LTO with WHOPR (gcc):

- Frontend parsing and IR generations are all done in parallel
- Backend compilations are done in parallel
- Inline decisions/analysis made in serial IPA
- Inline transformations in parallel within partitions during backend compilations
- Requires materializing clones into partitions, increasing serial IR I/O overhead
- Summary based IPA is done in **serial**
- Partitioning is done in **serial**
- IR reading and rewriting is done in **serial**

Parallel CMO: What if we can...

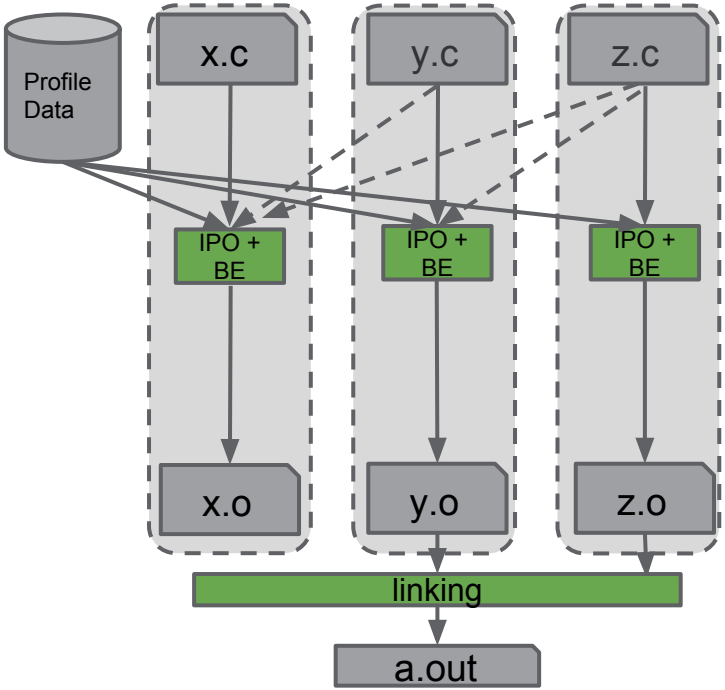


Fully Parallel CMO:

- End-2-end compilation in parallel
- Cross Module Optimization is enabled for each compilation
- Each module is extended with other modules as if they were from headers

(Note most of the performance comes from cross module inlining)

Lightweight IPO (LIPO) approach



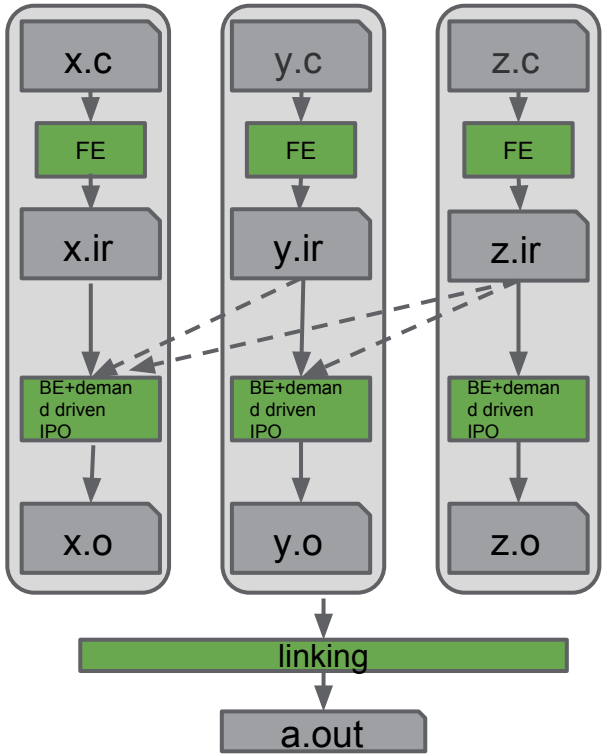
Fully Parallel CMO In LIPO mode (gcc/google):

- Source importing is pre-computed (using dynamic call graph from profile)
- Source compilations are extended with auxiliary modules during parsing
- Module groups are usually small or capped
- Captures most of the full LTO gain
- Compilations are fully parallel!

Problems with LIPO mode

- Need profile data to compute module group
- Importing done at module level limits the scalability and performance headroom
- Duplicated parsing/compilation of auxiliary modules
- Doesn’t tolerate stale profile

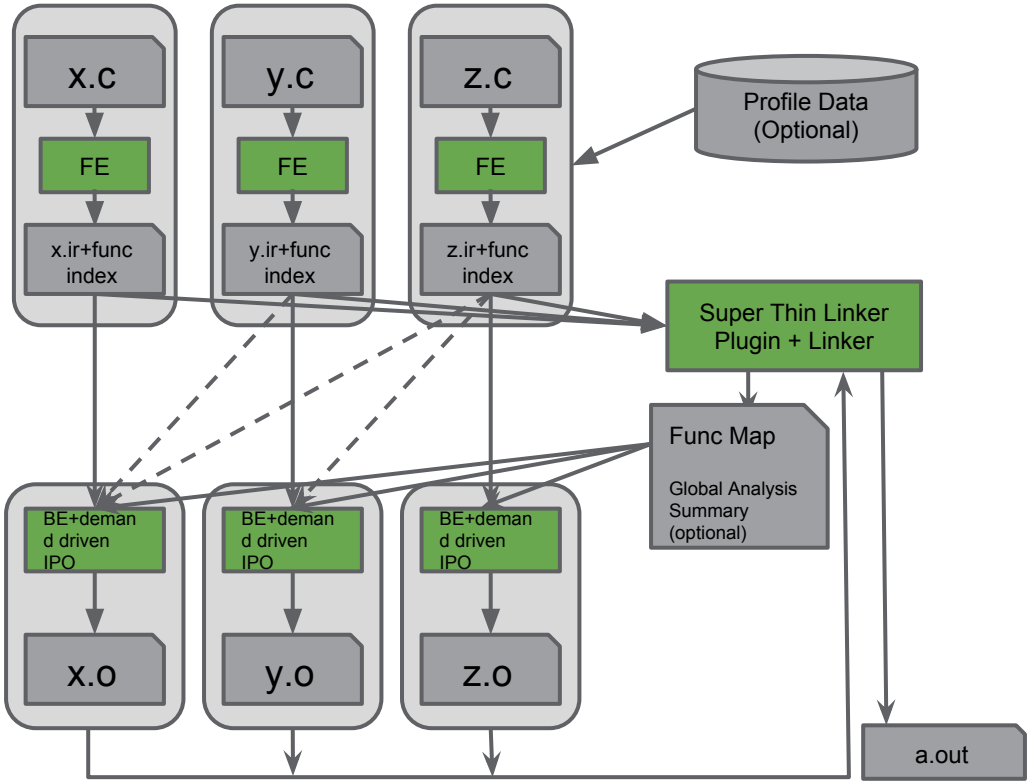
Full Parallel CMO: A new model



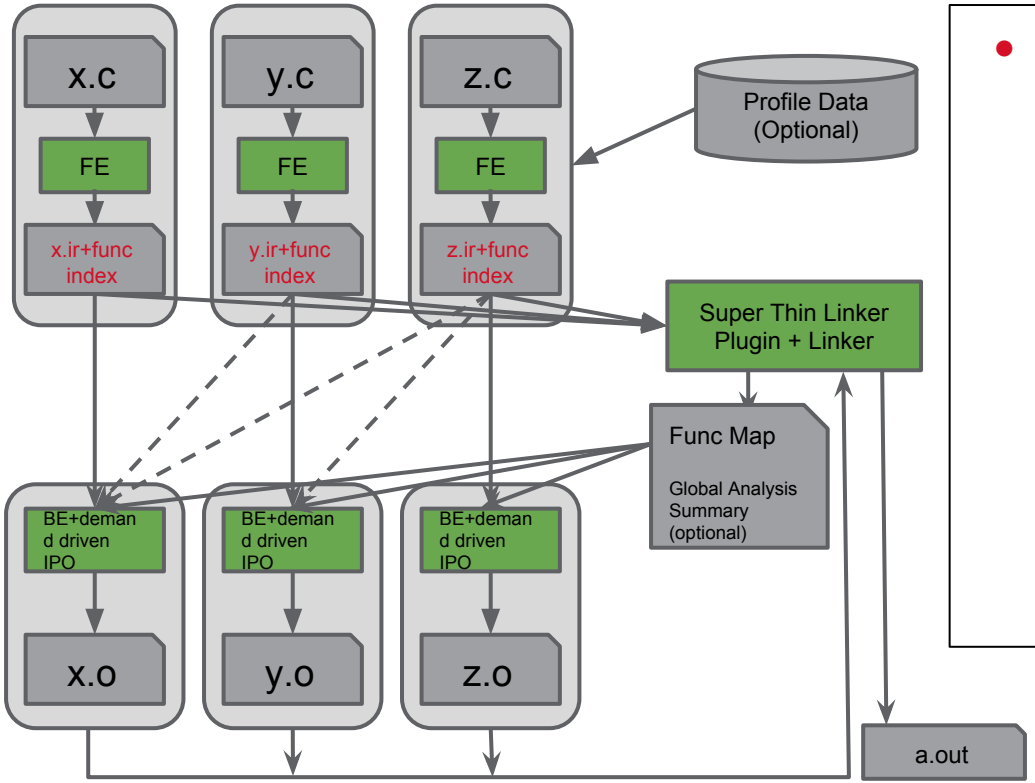
- Delay importing until IR files have been generated
- Allows fine grained importing at function level greatly increasing the number of useful functions that can be imported -- liberating more performance
- No more duplicated parsing in FE

(But how do they synchronize & communicate ?)

ThinLTO: An Implementation of the New Model

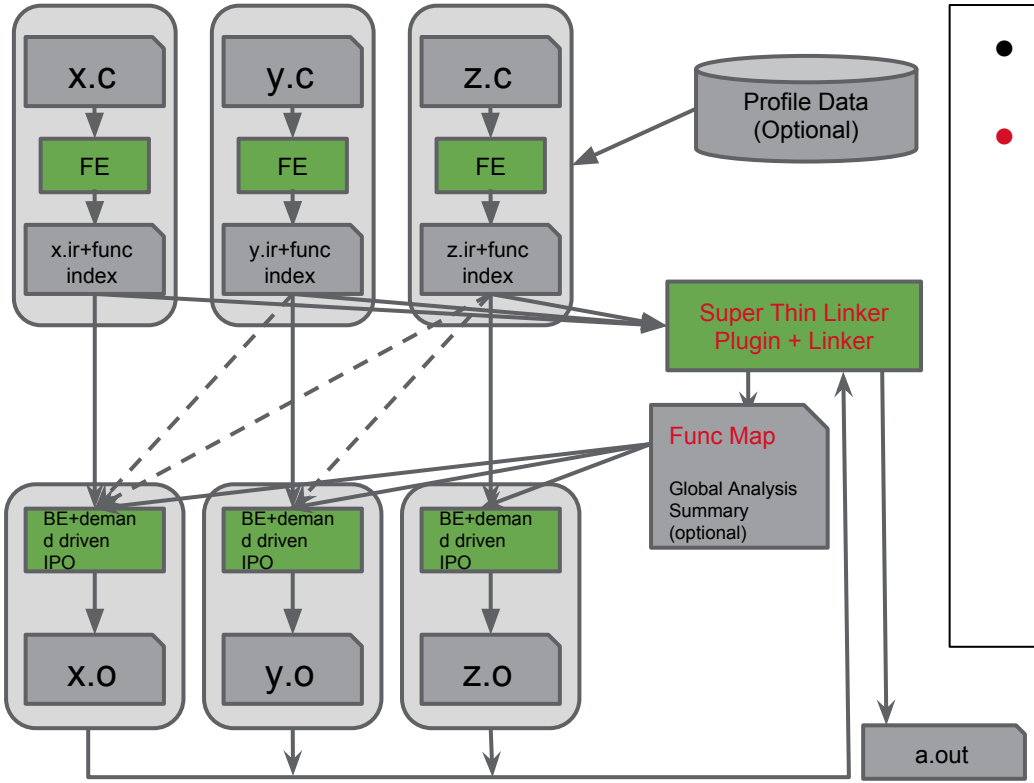


ThinLTO: An Implementation of the New Model



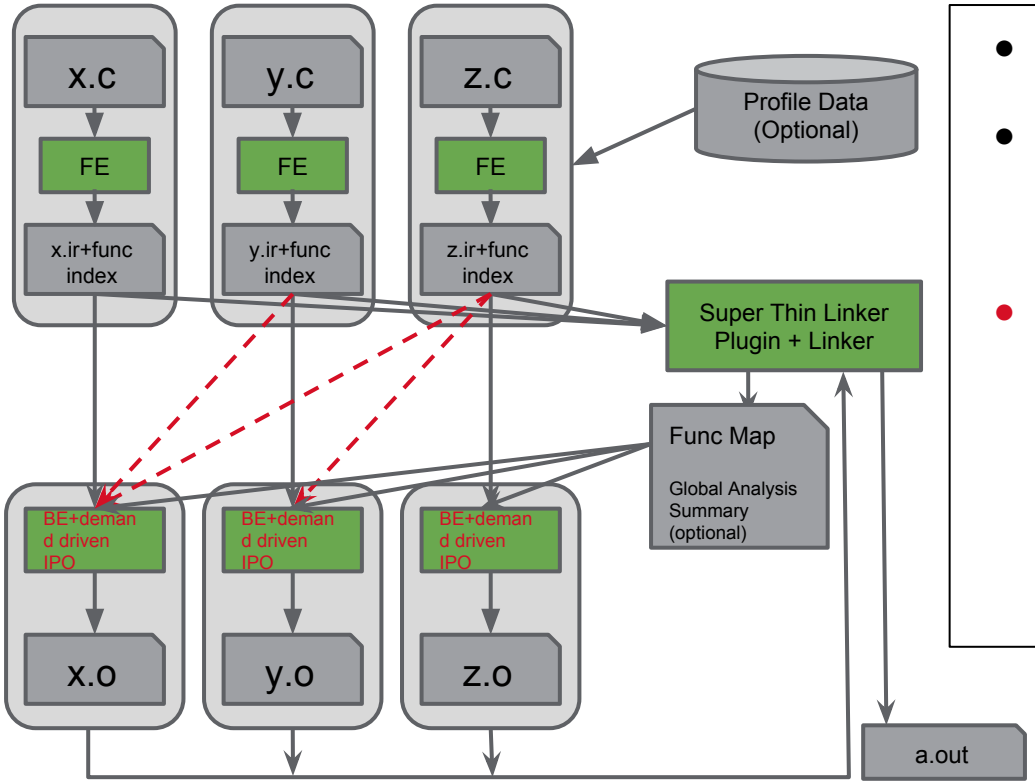
- IR includes summary data and function position index (can be in ELF symtab)

ThinLTO: An Implementation of the New Model



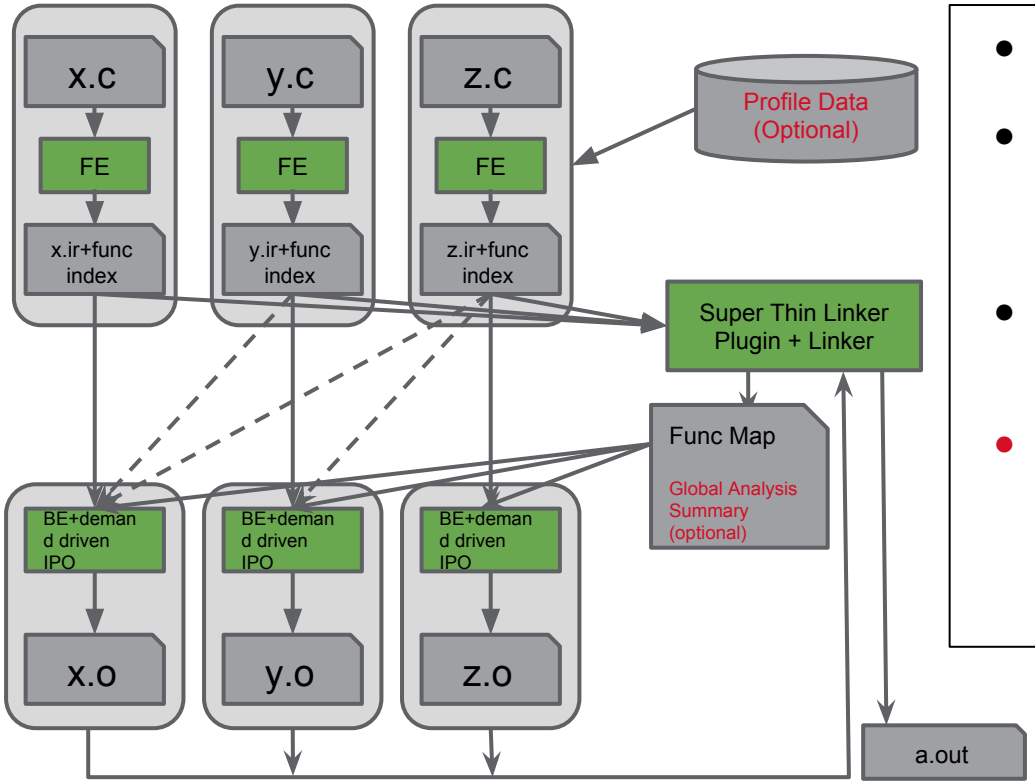
- IR includes summary data and function position index (can be in ELF symtab)
- To enable demand driven IPO in backend compilation, the ThinLTO plugin simply aggregates a global function map

ThinLTO: An Implementation of the New Model



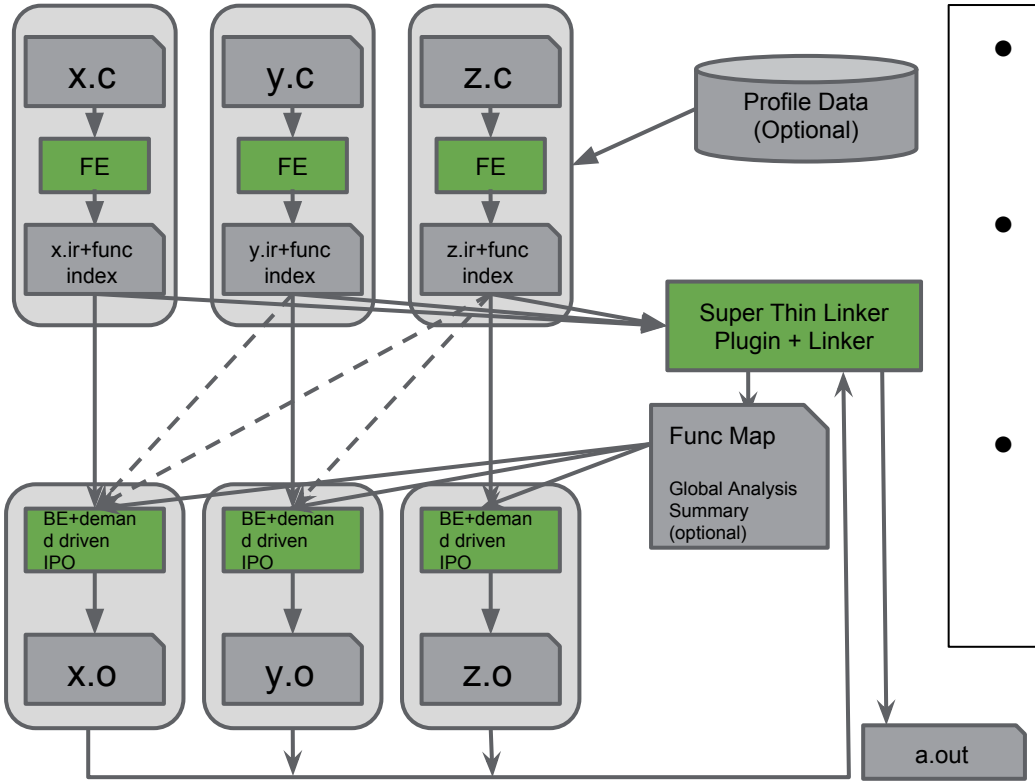
- IR includes summary data and function position index (can be in ELF symtab)
- To enable demand driven IPO in backend compilation, the ThinLTO plugin simply aggregates a global function map
- Enables backend to do importing at function granularity: minimizing memory footprint, IO/networking overhead

ThinLTO: An Implementation of the New Model



- IR includes summary data and function position index (can be in ELF symtab)
- To enable demand driven IPO in backend compilation, the ThinLTO plugin simply aggregates a global function map
- Enables backend to do importing at function granularity: minimizing memory footprint, IO/networking overhead
- Function importing is based on function summary, optional global analysis summary, and profile data, with a priority queue to maximize benefits

ThinLTO: An Implementation of the New Model



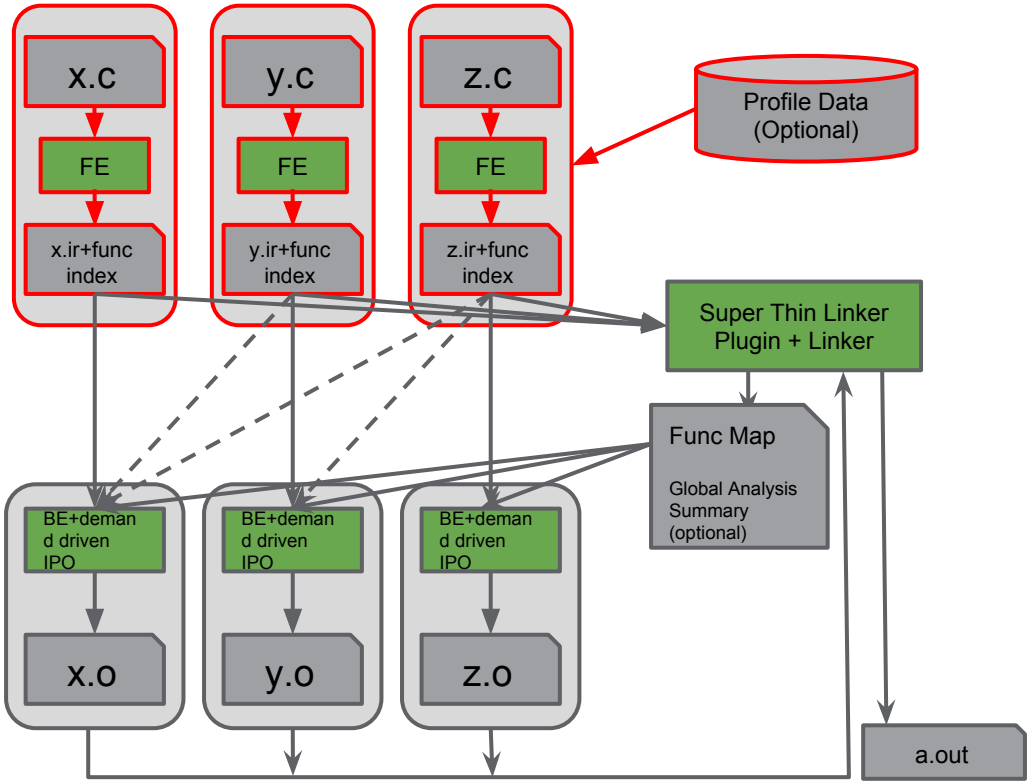
- ThinLTO plugin does very minimal work
  - No IPA by default
  - No IR reading, partitioning, and IR rewriting, so minimal I/O
- It can scale to programs of any size, and allow IPO on machines with tiny memory footprints and without significantly increasing time (requirements for enabling by default)
- For single node ThinLTO build, the BE parallel processes will be launched in the linker process by the plugin

ThinLTO Advantages

- Highly Scalable
  - Thin plugin layer does not require large memory and is extremely fast
  - Fully parallelizable backend
- Transparent
  - Similar to classic LTO, via linker plugin
  - Doesn’t require profile (unlike LIPO)
- High Performance
  - Close to full LTO
  - Peak optimization can use profile and/or more heavyweight IPA
- Flexible
  - Friendly to both single build machine and distributed build systems

➢ *Possible to enable by default at -O2!*

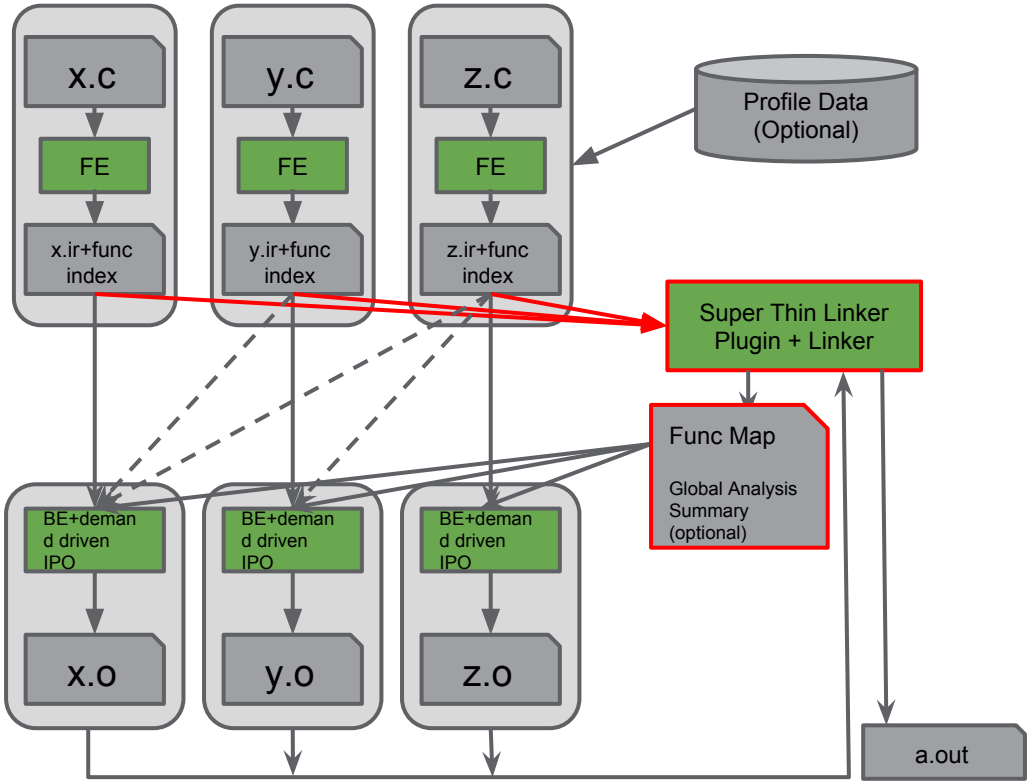
ThinLTO Phase 1: IR and Function Summary Generation



ThinLTO Phase 1: IR and Function Summary Generation

- Generate IR files in parallel
  - E.g. bitcode as in a normal LLVM -flto -c compile
- Generate function index table
  - Maps functions to their offsets in the bitcode file
- Generate function summary data to guide later import decisions
  - Included in the function index table
  - E.g. function attributes such as size, branch count, etc
  - Optional profile data summary (e.g. function hotness)
- How to represent function index/summary table?
  - Metadata? LLVM IR?
  - ELF section? Discussed later...

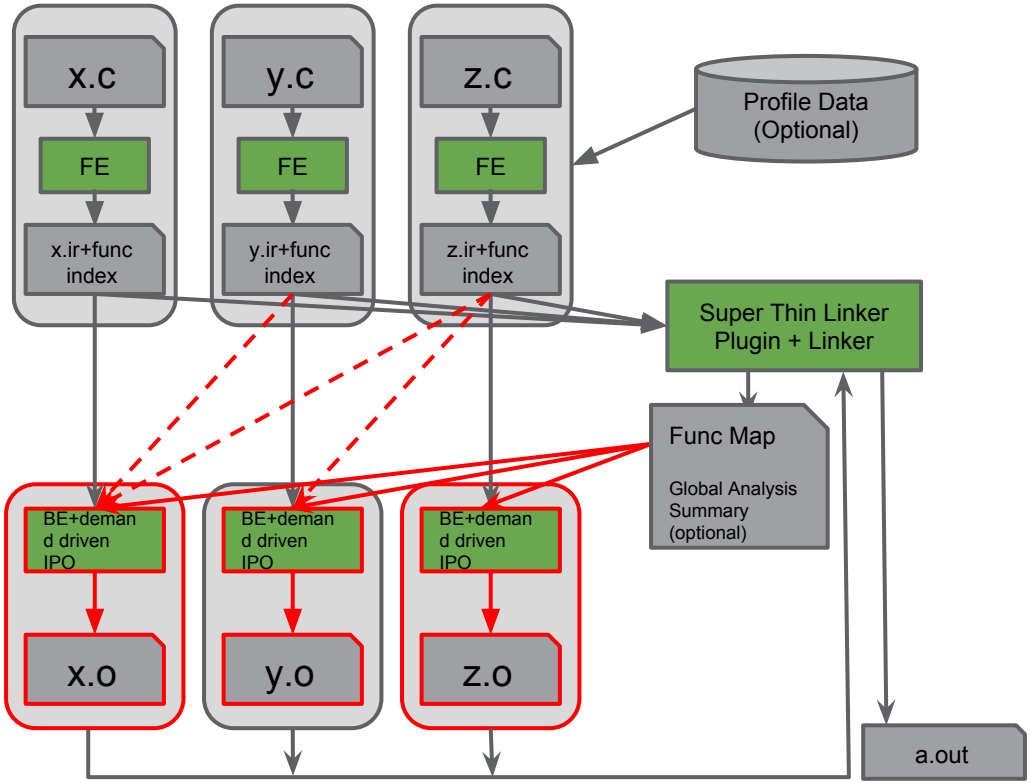
ThinLTO Phase 2: Thin Linker Plugin Layer



ThinLTO Phase 2: Thin Linker Plugin Layer

- Launched transparently via linker plugin (as with LTO)
- Simply combine per-module function indexes into a thin archive
  - On disk hash table or AR format if function indexes in ELF
- Optional heavier-weight IPA added for (non-default) peak optimization levels and saved in the combined summary
- Exclude functions from map if unlikely to benefit from inlining
  - E.g. very large, unlikely/cold, duplicate COMDAT copies, etc
  - Can aggressively prune import candidates (In LIPO only ~5-10% of functions in hot imported modules are actually needed)
- Generate backend compile Makefile
  - For single node build invoke parallel make and resume final link

ThinLTO Phase 3: Parallel BE with Demand Driven IPO



ThinLTO Phase 3: Parallel BE with Demand Driven IPO

- Iterative lazy function importing
  - Priority determined by summary and callsite context
  - Use index in combined function map to import function efficiently
- Global value symbol linking as functions imported
- Static promotion and renaming (similar to LIPO)
  - Uniqued names after linking must be consistent across modules
  - Minimize required promotions with careful import strategy
- Lazy debug import
- IPA/CMO on extended module
  - Afterwards discard non-inlined imported functions, except referenced COMDAT and referenced non-promoted statics
- Late global and machine specific optimizations, code generation

ThinLTO Function Import Decisions

- Ideally import a close superset of functions that profitable to inline
  - Use minimal summary information and callsite context to estimate
  - Achieve some of LTO's internalization benefit by importing functions with a single static callsite as per summary (i.e. call once local linkage inline benefit heuristic)
  - More accurate with optional profile data (particularly with indirect call profiles enabling promotion in the first stage compile).
- Minimize required static promotions
  - Always import referenced statics, so promote only address-taken statics (i.e. may still be referenced by another exported function)

IR and Function Map Format

Two main options for IR and function index/summary map:

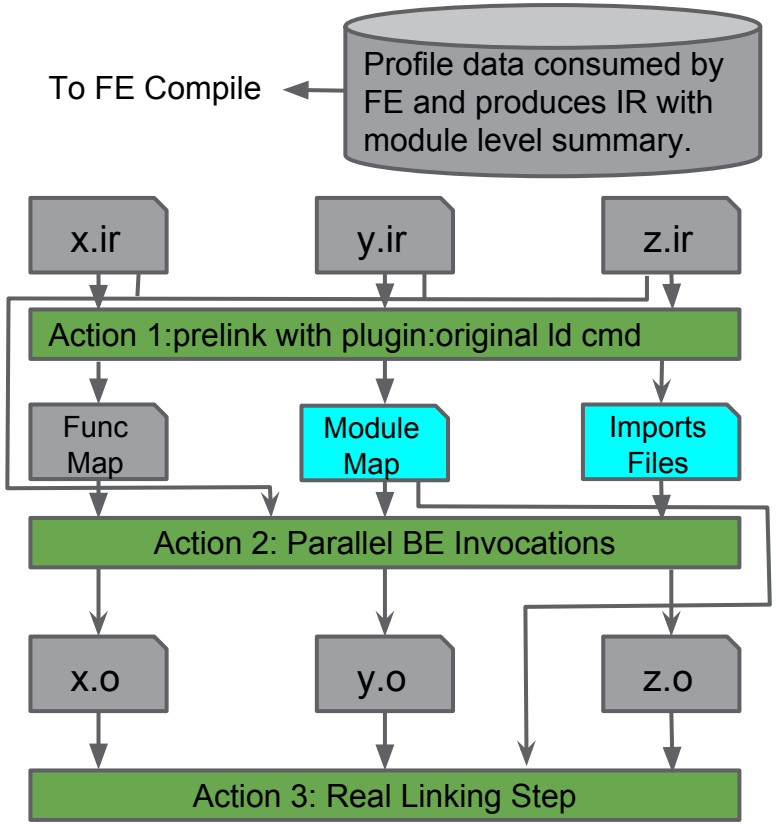
1. Bitcode
   - Stage 1 per-module function map represented with new metadata
   - Could encode combined function map as an on disk hash table (ala profile)
   - Tools like $AR, $RANLIB, $NM and “$LD -r” must be invoked with a plugin to handle IR
2. ELF wrapper
   - Leverage recent support for handling ELF-wrapped LLVM bitcode
   - Stage 1 per-module function maps encoded in special ELF section
   - Combined function map ELF sections can simply use $AR format
   - Tools like $AR, $RANLIB, $NM and “$LD -r” just work

The transparency with standard tools is a big advantage of using an ELF wrapper, but encoding the function map as metadata is the simplest/fastest route to implementing within LLVM. Looking for feedback from the community on this aspect.

ThinLTO Distributed Build Integration: Challenges

- How/where should the BE jobs be distributed/dispatched to the build cluster
- BE build input sets preparation/staging:
  - ThinLTO works best when build nodes share network file system. Lazy importing can minimize network traffic needed for CMO
  - Otherwise BE compile dependency needs to be precomputed for file staging from local disks or network storage (compute in plugin layer from profile data, or from symbol table and heuristics at O2)
- Incremental builds:
  - Incremental compile for IR files works with any build system
  - Using BE compile dependency list, the BE compilation can be incremental as well

ThinLTO Distributed Build Integration: Example



Split the ThinLTO link step into 3 actions:

1. A pre-link step with ThinLTO plugin producing global data:
   a. Function Map, Module Map (from IR to real object), Imports Files (for file staging)
   b. Exits without producing real objects
2. Backend invocation action
   a. Uses map files from step 1 to create backend command lines and schedule jobs
3. Real linking
   a. Fix up the original command line using the module map and perform real link

LLVM Prototype Status

- Implemented prototype within LLVM
- Support for stage 2 (thin plugin layer) and 3 (BE with importing) in both gold plugin and llvm-lto driver under options
- Function map/summary and thin archive in separate side text files for now
- Testing with SPEC cpu2006 C/C++ benchmarks

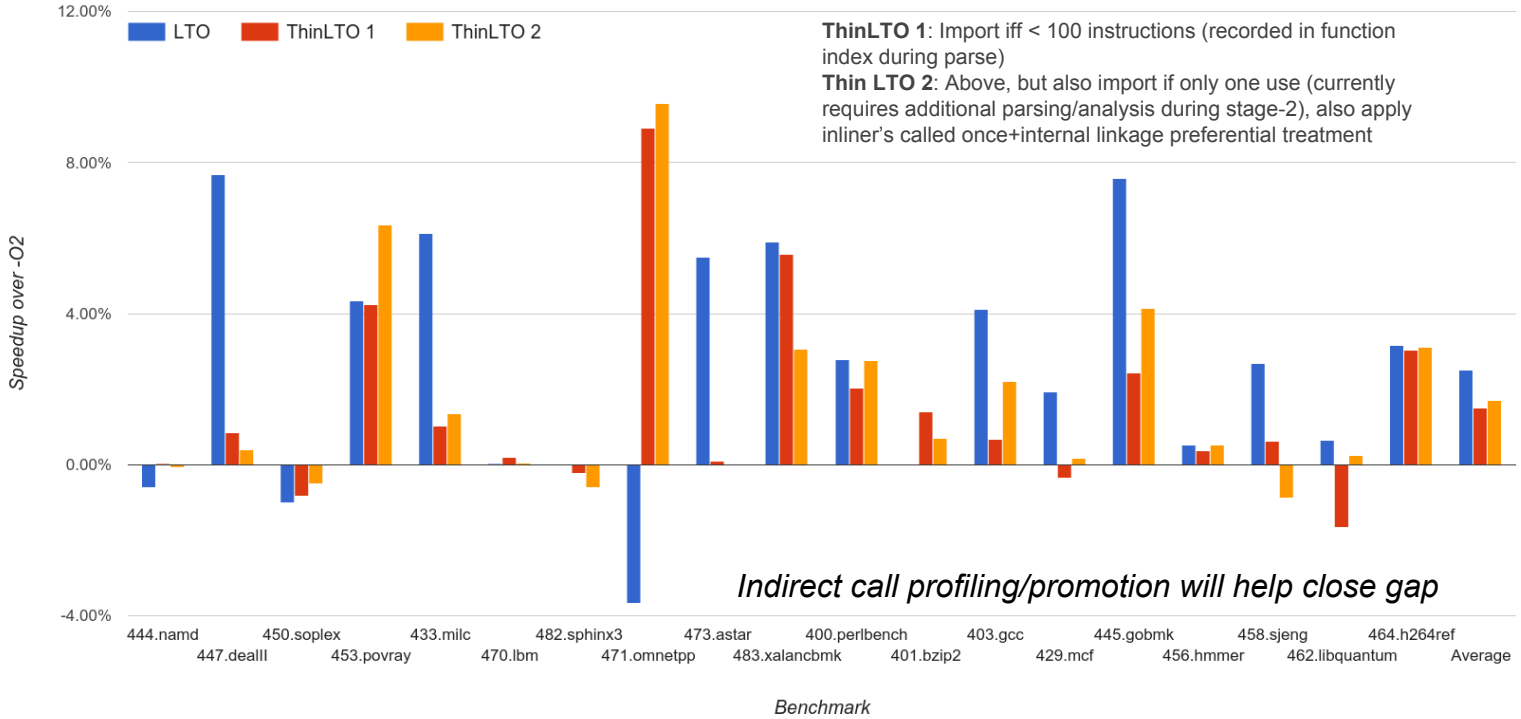
LLVM Prototype Details

- ThinLTO importing implemented as SCC pass before inlining
  - Leverage LTO module linking, with ThinLTO-specific modifications
- Initial heuristics for guiding import decisions (e.g. function size, profile, call count from function summary)
- Implemented minimized static promotion
- Rename statics by appending MD5 hash of module name/path
- Enhanced GlobalDCE to remove unneeded imported functions
- Lazy debug import as a post-pass during ThinLTO pass finalization
  - Only import DISubroutine metadata needed by imported functions
  - Uses bookkeeping to track temporary metadata on imported instructions to enable fixup as debug metadata imported/linked

Preliminary Experimental Data

- Very little tuning, preliminary import heuristics
  - Limit the number of instructions (computed at parse time)
  - Try allowing more aggressive import/inline when single use
  - No results with profile data yet
  - ThinLTO negatively impacted by lack of indirect call visibility (needs to be addressed with value profile/promotion during stage-1)
- Runtime system configuration:
  - Intel Corei7 6-core hyperthreaded CPU
  - 32K L1I, 32K L1D, 256K L2, 12M shared L3
- SPEC CPU2006 C/C++ benchmarks
  - Averaged the results across 3 runs

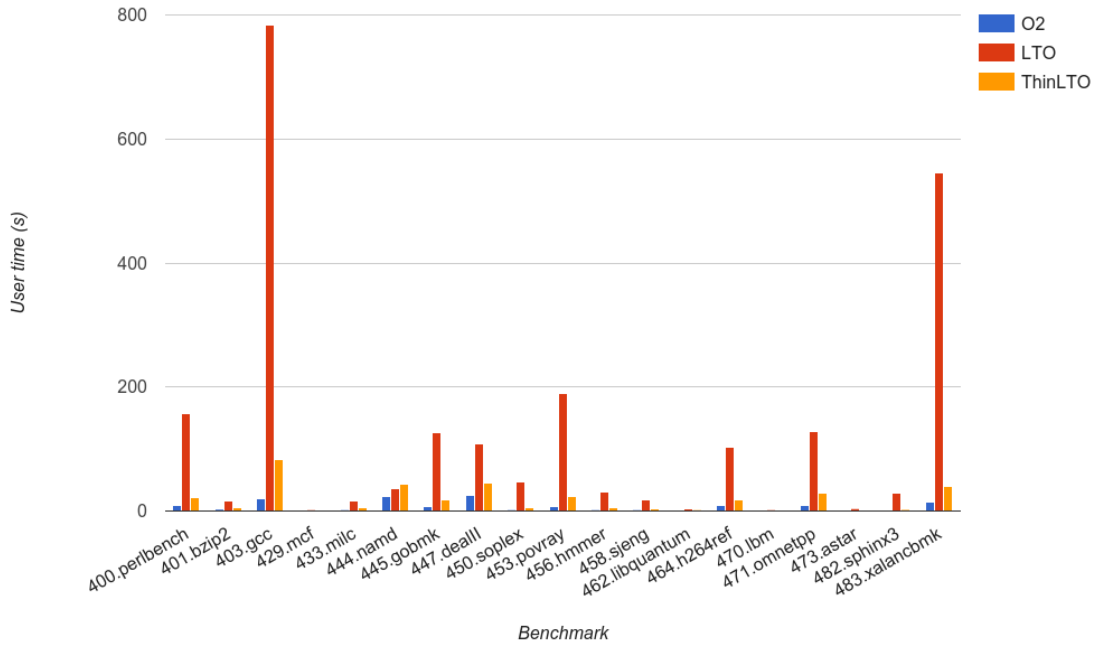
Performance comparison



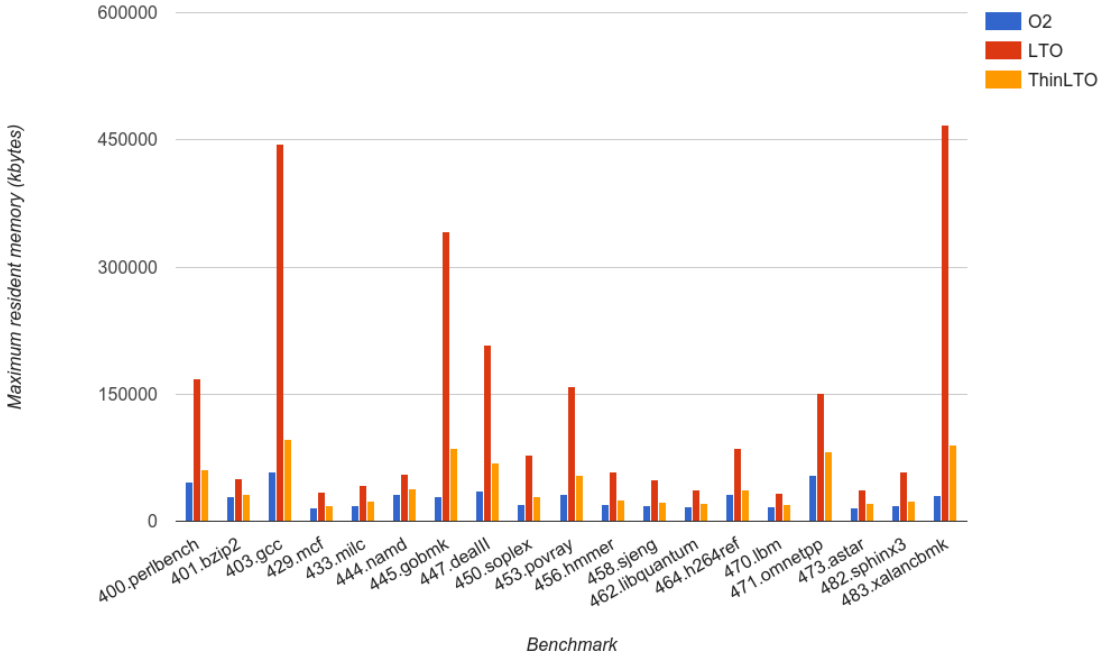
Build Overhead Comparisons

- Build system configuration:
  - Dual Intel 10-Core Xeon CPU
  - 64GB Quad-Channel
- SPEC CPU2006 C/C++ benchmarks
  - Small compared to many real-world C++ applications!
- Compare maximum LTO+BE times and memory
  - Exclude parsing to compare optimization time
    - All use the same “clang -O2 -flto -c” bitcode files as input
  - For ThinLTO: Max BE (including ThinLTO importing) time/memory
    - For a distributed build the BE jobs will be in parallel
  - For O2: Max optimization time/memory measured with llc -O2

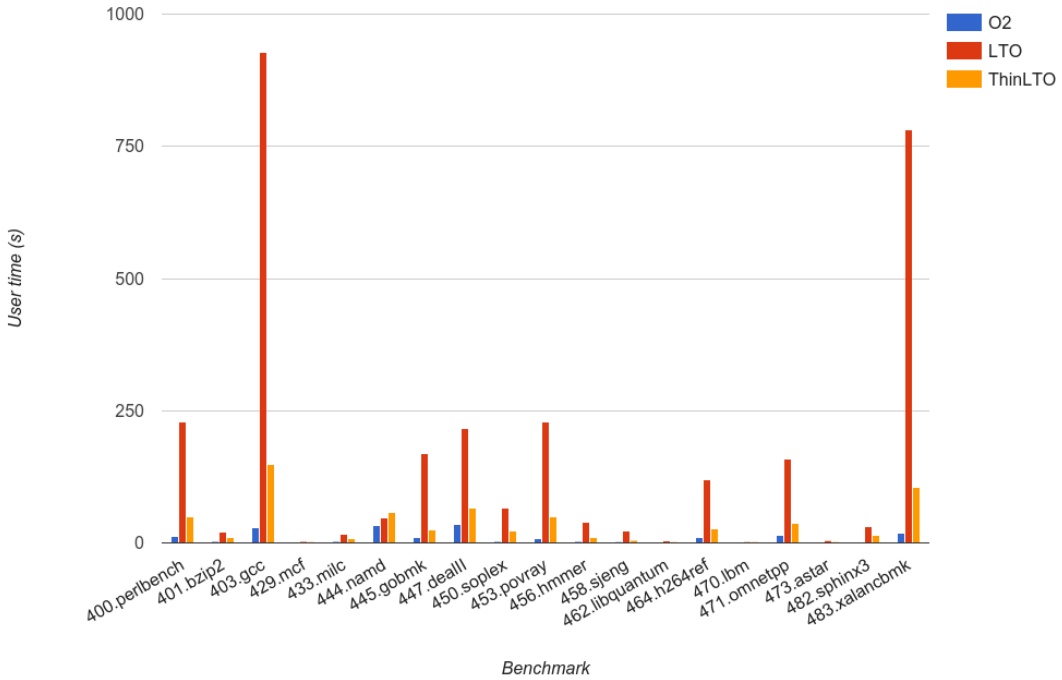
Compile time comparison (no -g)



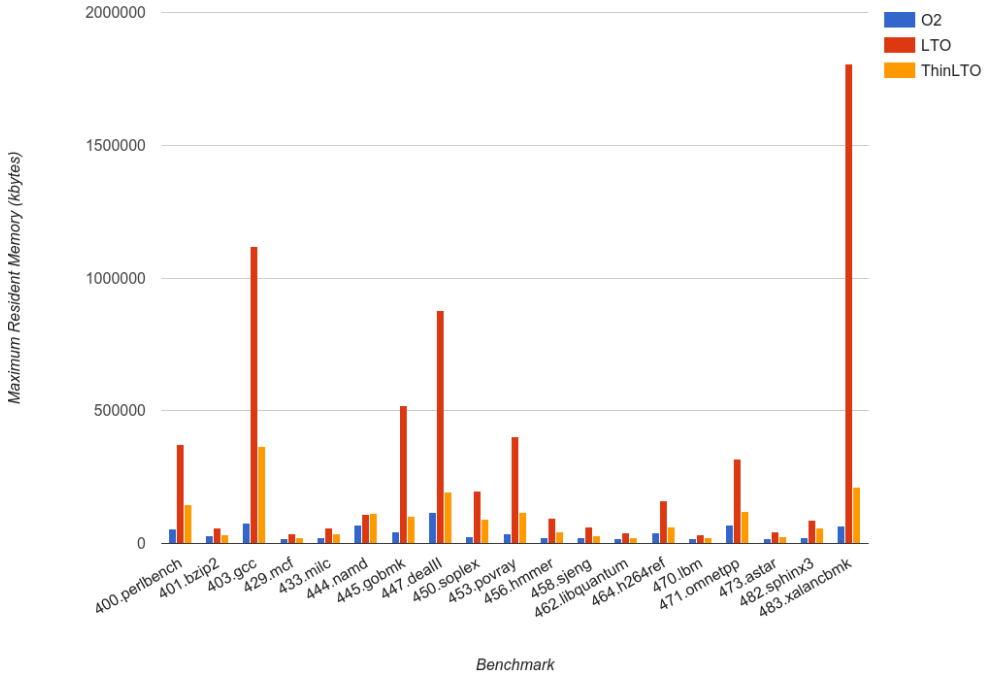
Memory Comparison (no -g)



Compile time comparison (-g)



Memory Comparison (-g)



Next Steps

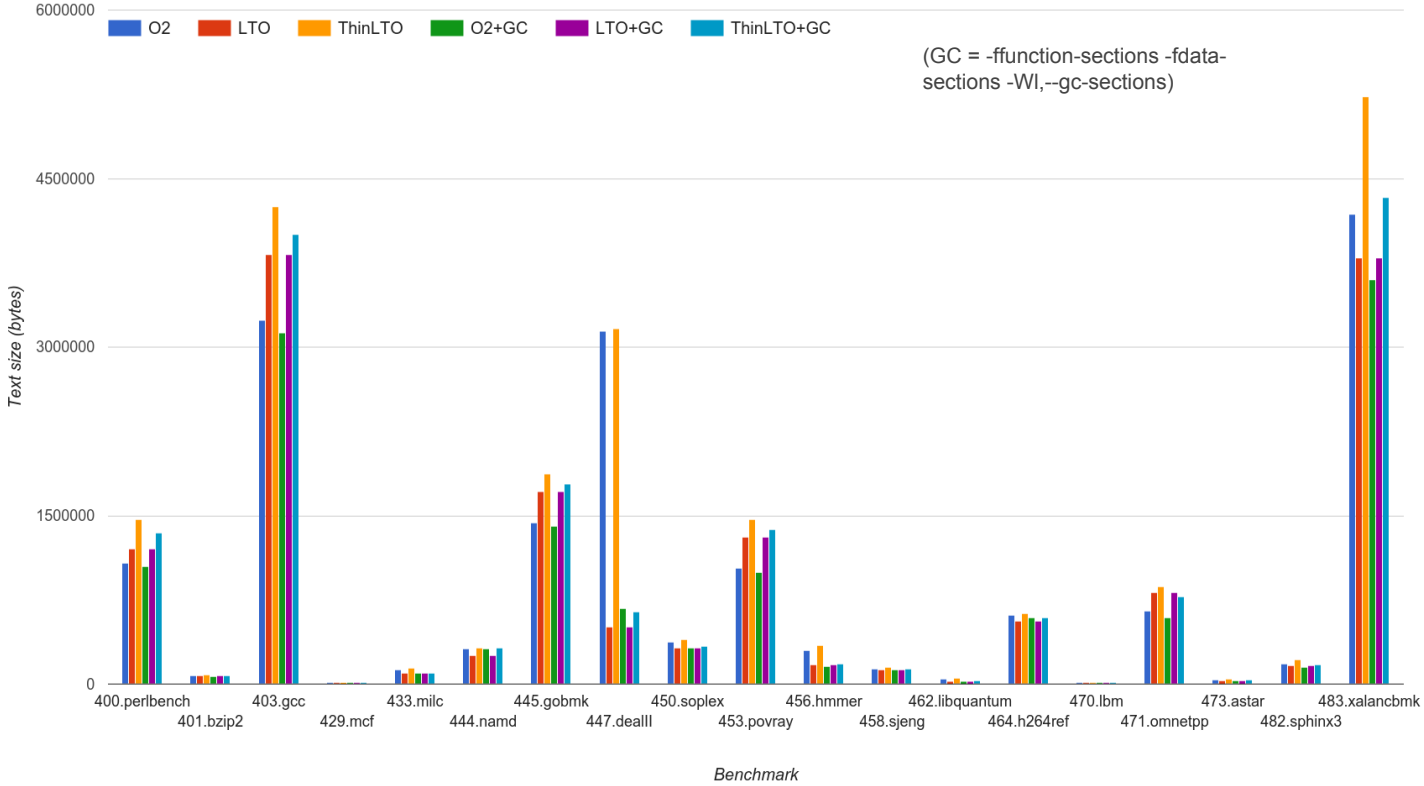
- Decide on file formats
- Fine-tune import heuristics
- Profile feedback integration
- Distributed build system integration
- Decide how to best stage upstream patches

Thank you!

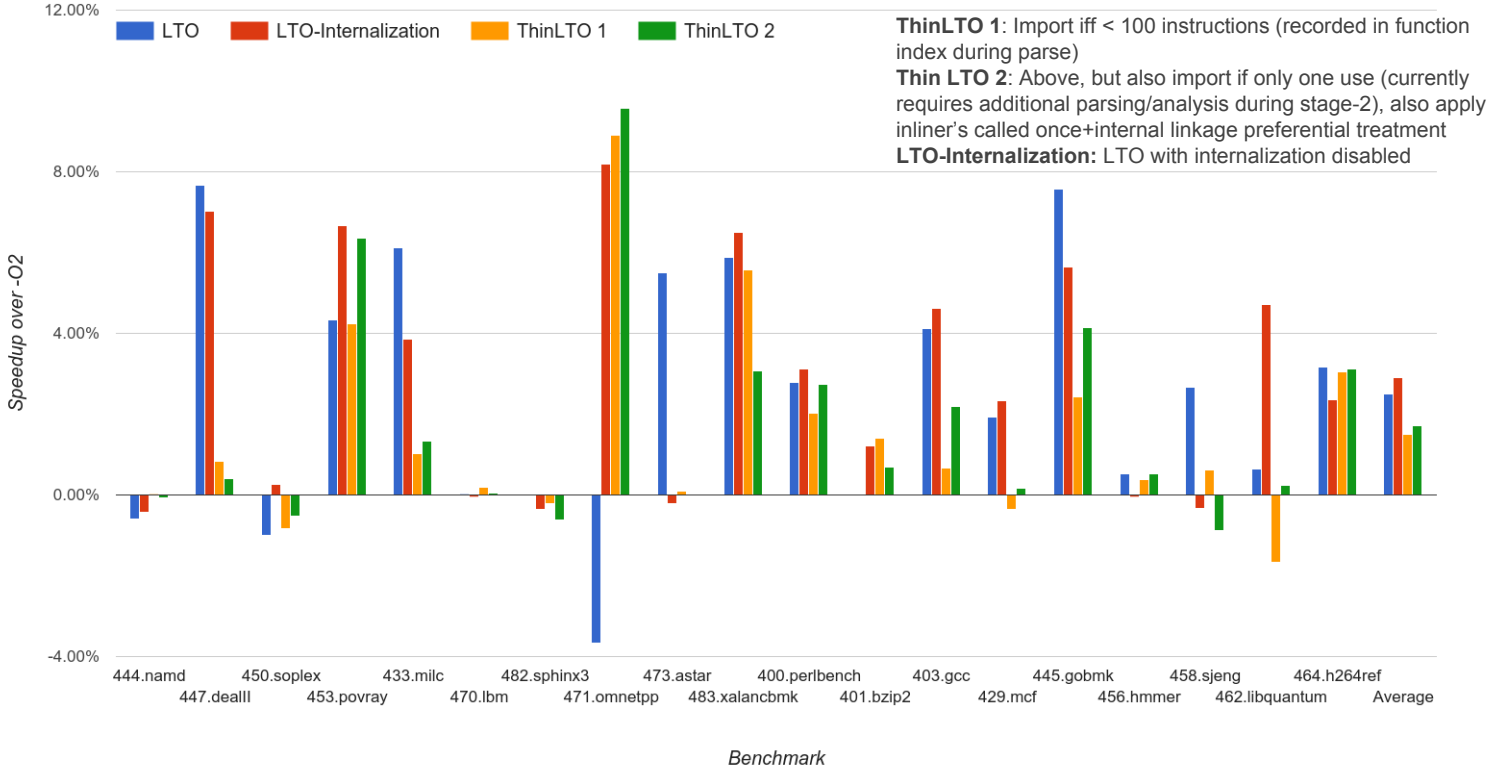
Teresa Johnson (tejohnson@google.com)
Xinliang David Li (davidxl@google.com)

Backup

Text Size Comparison



Performance Effect of Internalization



ThinLTO 1: Import iff < 100 instructions (recorded in function index during parse)
Thin LTO 2: Above, but also import if only one use (currently requires additional parsing/analysis during stage-2), also apply inliner's called once+internal linkage preferential treatment
LTO-Internalization: LTO with internalization disabled