ThinLTO Summaries for Incremental JIT Compilation

Stefan Gränitz
Freelance Dev C++ / Compilers / Audio

LLVM Social Berlin
31. Mai 2018

LTO in a Nutshell

- LTO = Link Time Opt. = Whole Program Opt.
  = Cross-Translation-Unit Optimization = Monolithic LTO
- Compilers can only inline code that they see, e.g.:
  C++ definitions from included headers, but not from cpp's
- Linkers can have the full picture, but traditional LTO is expensive and hard to scale

LTO in a Nutshell

- LTO = Link Time Opt. = Whole Program Opt.
  = Cross-Translation-Unit Optimization = Monolithic LTO
- Compilers can only inline code that they see, e.g.:
  C++ definitions from included headers, b… t from cpp's
- Linkers can have the full picture, …
  expensive and hard to scale

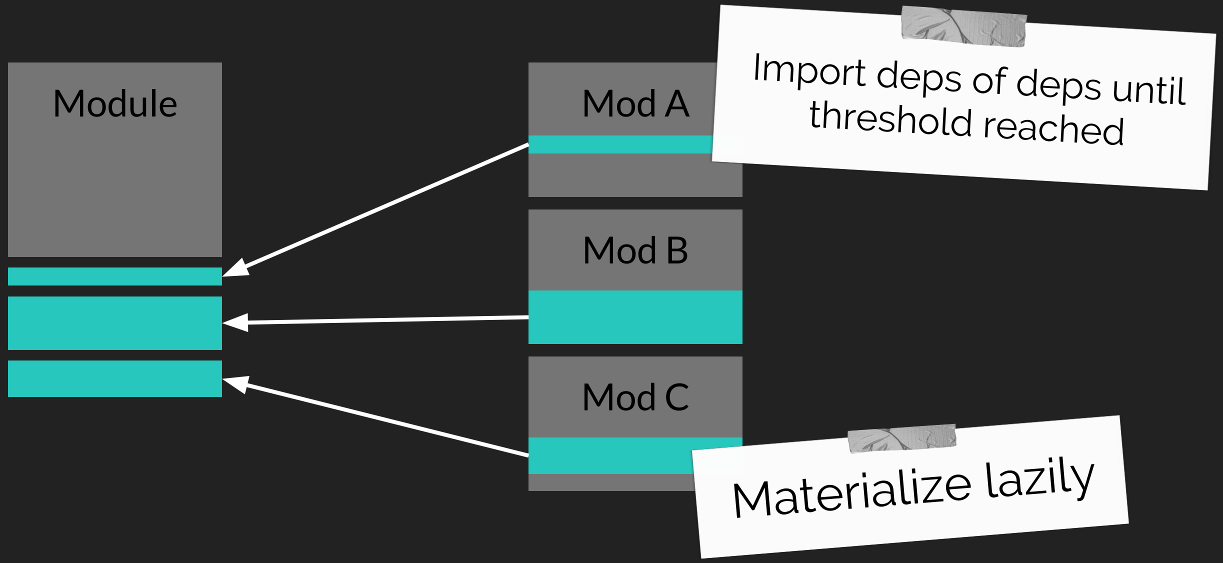
Requires IR code in object files
https://llvm.org/docs/BitCodeFormat.html#native-object-file-wrapper-format

ThinLTO in a Nutshell

- Compiler emits summaries with call graph info to object files
- Linker reads all summaries and builds global combined index for Interprocedural Analysis (Thin-Link) based on GUIDs
- Then runs ThinLTO Backend per module:
  - Import function definitions from other modules
  - Execute Interprocedural Transformations

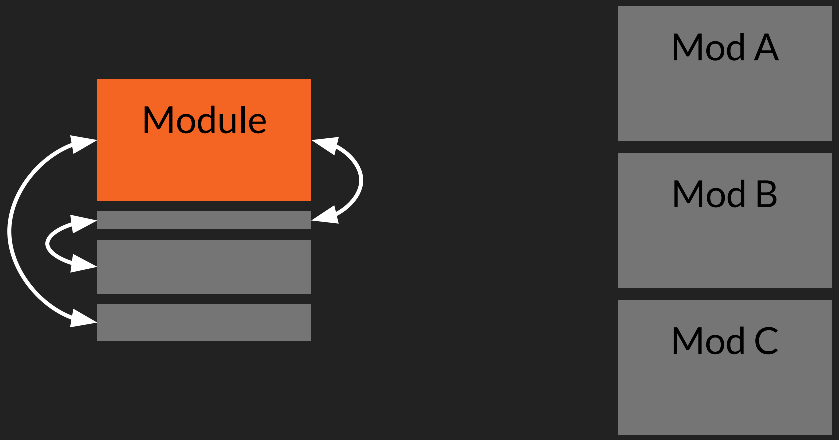
ThinLTO Backends

→ 1. Import functions + dependencies from other modules



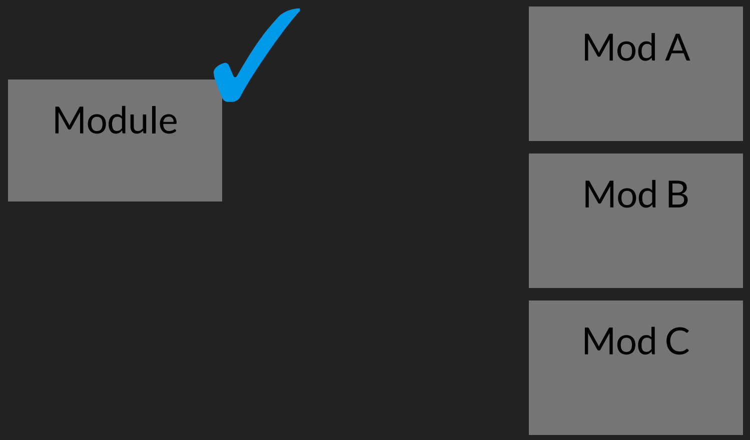
ThinLTO Backends

→ 2. Run inlining and IPO



ThinLTO Backends

→ 3. Drop imports



ThinLTO Benefits

- Small overhead: compact summaries, fast combining
- Good scalability: actual transformations run in parallel
- Incremental: simple object file caching + your linker's magic
- Speedup: close to Monolithic LTO
- Development: less manual optimization avoids discussions on where to put a definition and can reduce review times dramatically :-)

ThinLTO Command Line

- Compile:

```
$ clang -flto=thin -O2 file1.c file2.c -c
```

- Link:

```
$ clang -flto=thin -O2 file1.o file2.o -o a.out
```

- All in one:

```
$ clang -flto=thin -O2 file1.c file2.c -o a.out
```

More Info

ThinLTO: Scalable and Incremental Link-Time Optimization

Teresa Johnson @ CppCon 2017

https://www.youtube.com/watch?v=p9nH2vZ2mNo

ThinLTO: Scalable and Incremental LTO

Teresa Johnson, LLVM Project Blog 2016

http://blog.llvm.org/2016/06/thinlto-scalable-and-incremental-lto.html

Clang ThinLTO Documentation

https://clang.llvm.org/docs/ThinLTO.html

Incremental Compilation in a Nutshell

- → Rebuild only what's necessary
- → Terminology:
  - Incremental Linking ≈ translation unit granularity
  - Incremental Compilation = higher gran. / smaller entities
- → Complexity in languages with C-like translation units:

  Incremental Linking << Incremental Compilation

Incremental Compilation in a Nutshell

- Rebuild only what's necessary
- Terminology:
  - Incremental Linking ≈ translation unit granularity
  - Incremental Compilation = higher gran. / smaller entities
- Con ... with C-like ...

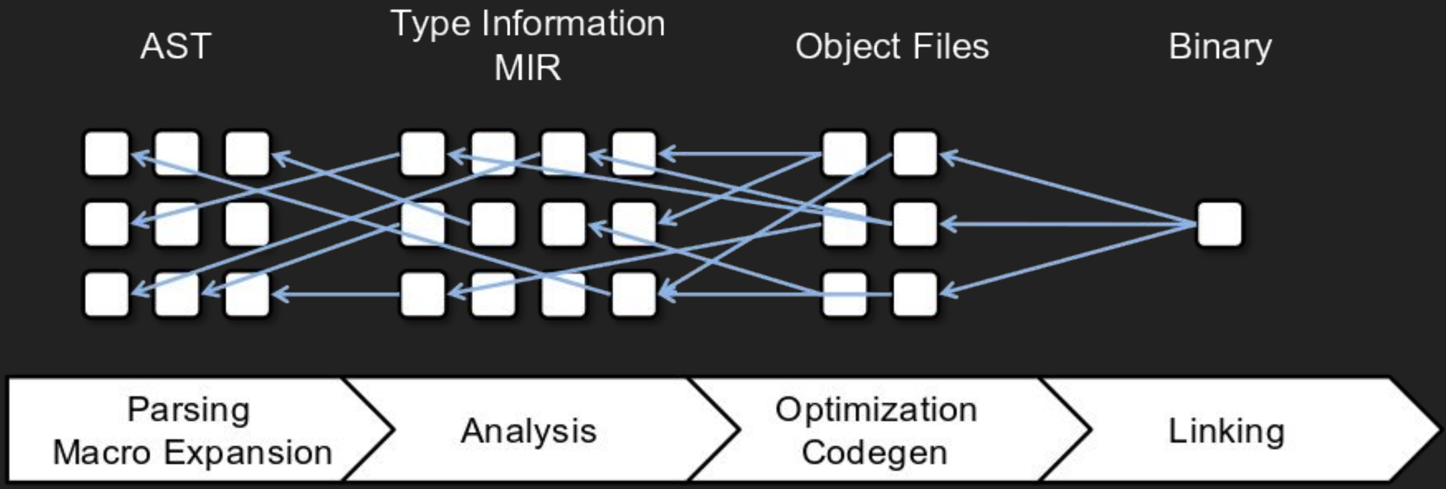
  Incr ... cremental ...

Dynamic linking to ↓turnaround times! e.g. libLLVM

#Entities↑ ⇒ Overhead↑
⇒ Eats up benefits

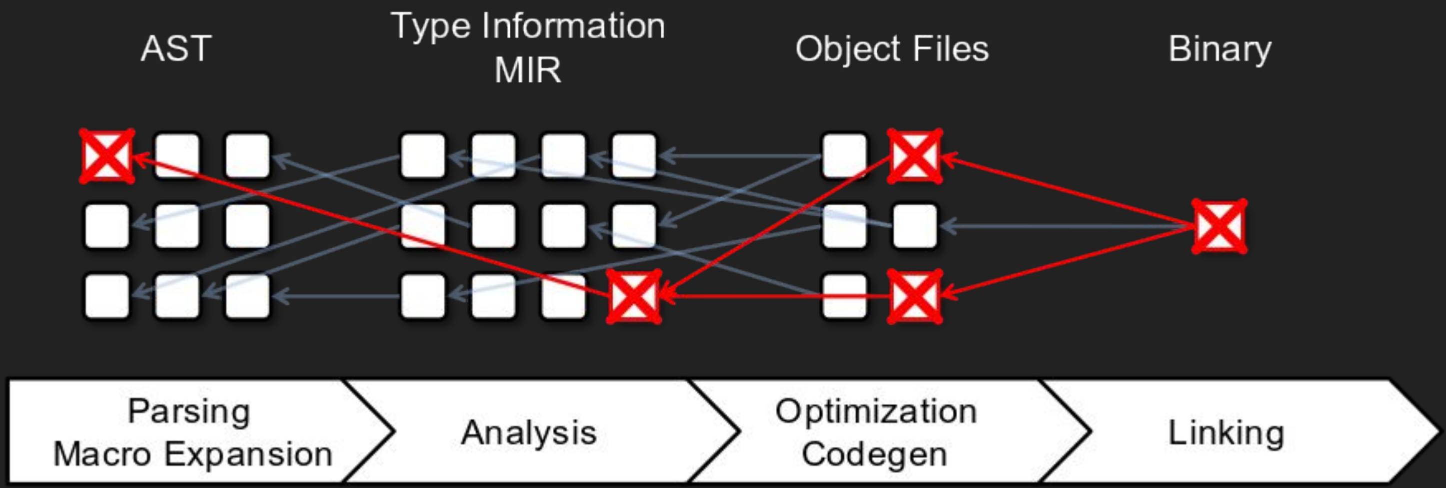
Incremental Compilation in the rustc Compiler

- AST node granularity



Incremental Compilation in the rustc Compiler

→ AST node granularity gets complex quickly



Incremental Compilation in the rustc Compiler

- → AST node granularity gets complex quickly
- → Compile-time savings hard to judge: “Up to 15% or More” *
- → Conceptually promising, kind-of-academic research

More Info

Incremental Compilation

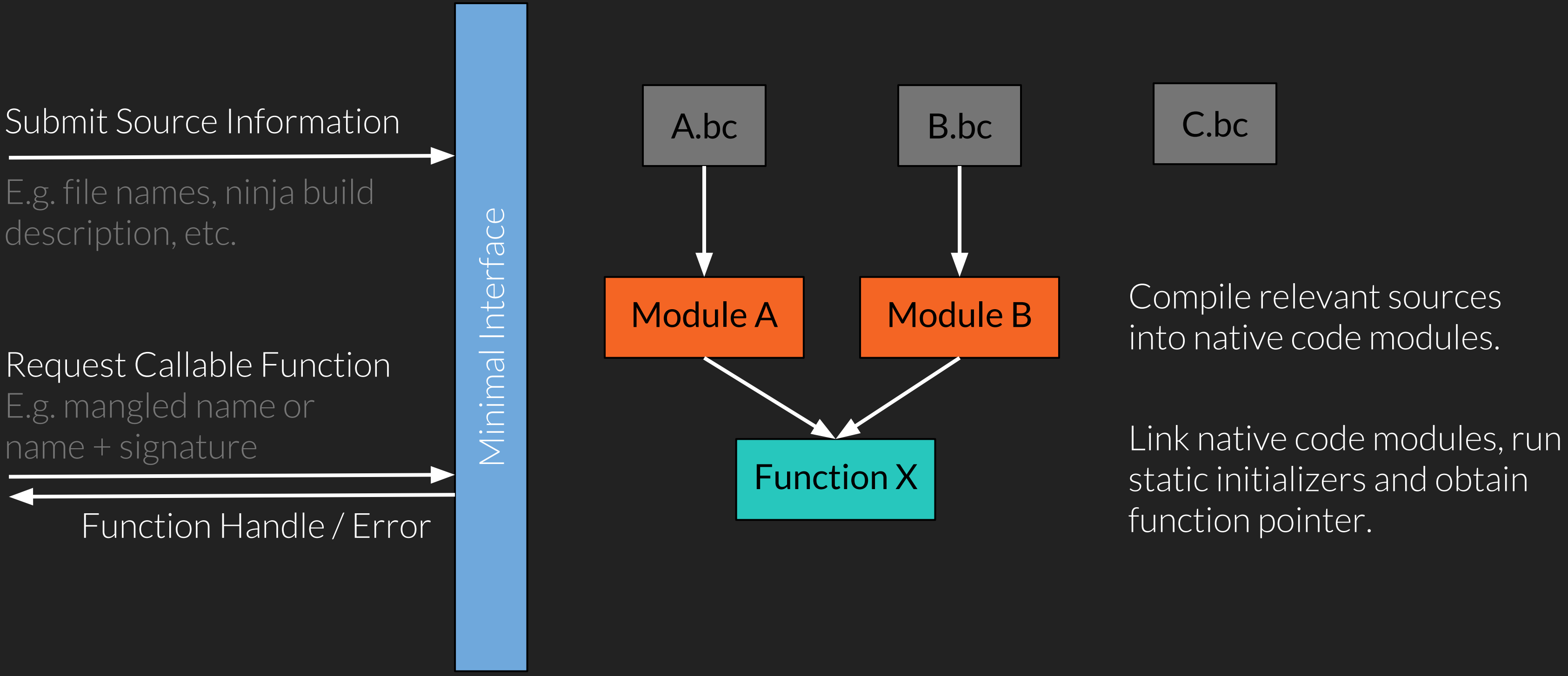
Michael Woerister, Rust Programming Language Blog 2016

https://blog.rust-lang.org/2016/09/08/incremental.html

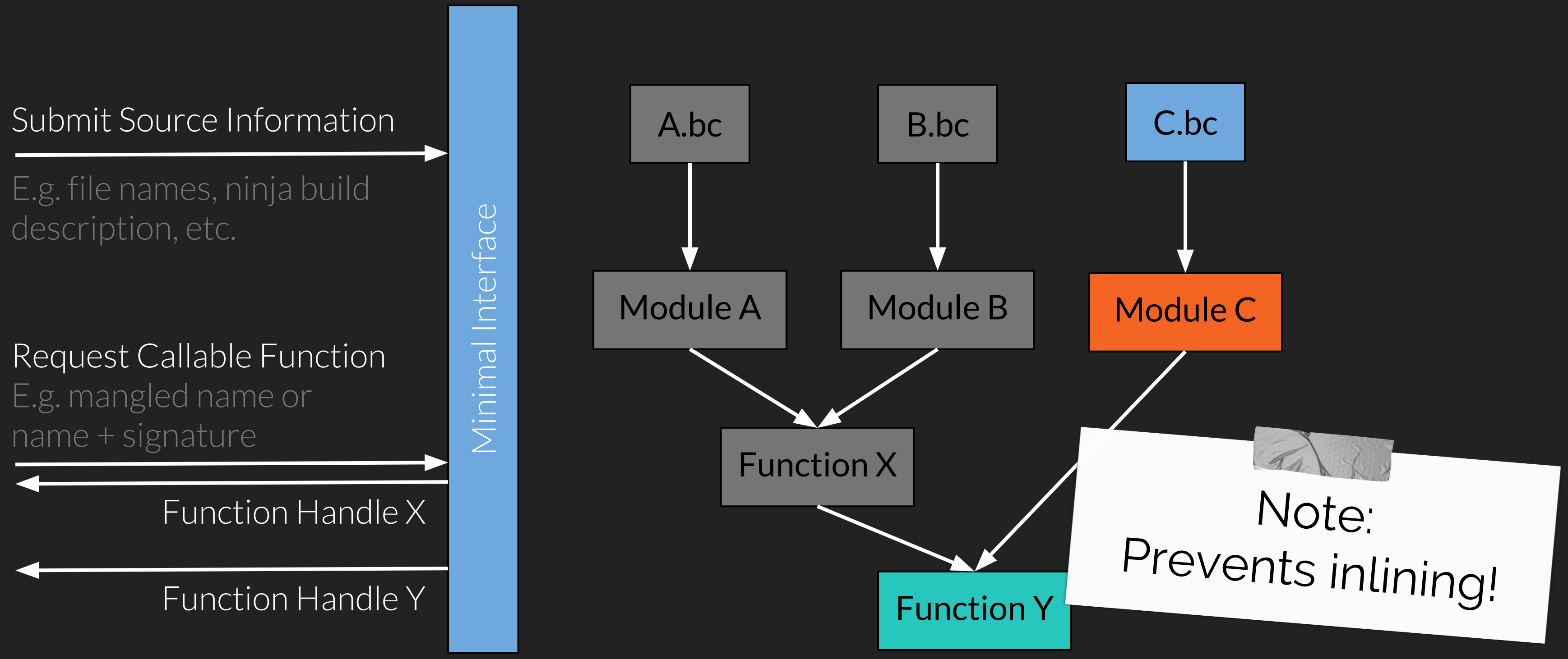
JIT Compilation

- JIT Compiler does both, compile and link
  - ↪ Source representation is LLVM IR
  - ↪ Incremental Compilation ≈ Incremental Linking
    No simple separation

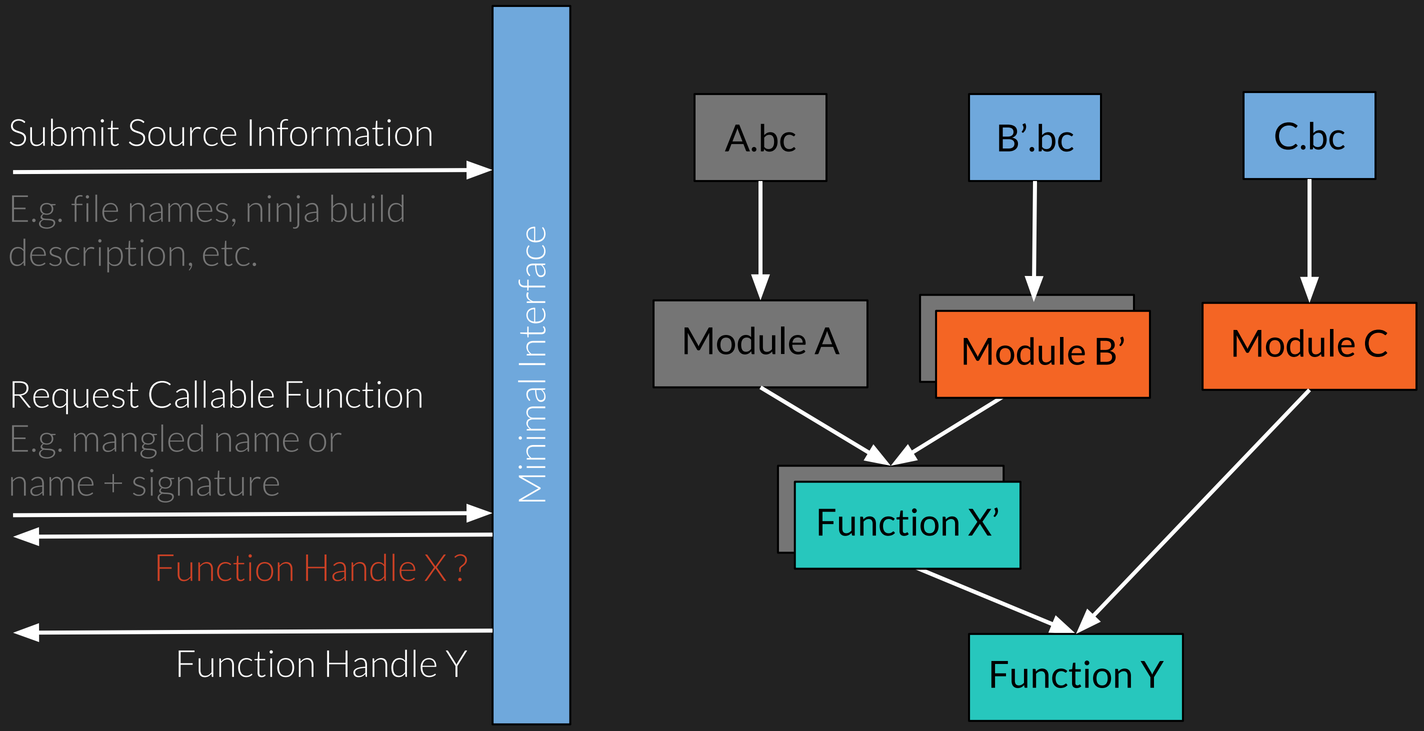
“Monolithic” JIT Compilation



Incremental JIT Compilation



Incremental Just-in-Time Compilation



Incremental JIT Compilation

ThinLTO Summaries help

- Requires quite some housekeeping
  - ↪ Track file changes on disk — Source Module Hashes
  - ↪ Determine source dependencies for functions — Call Graph
  - ↪ Decide what needs to be recompiled
  - ↪ Link against existing code
  - ↪ Determine static initializers to rerun
  - ↪ Invalidate existing function handles (and rebind?)

Incremental JIT Compilation

- Requires quite some housekeeping

- More importantly we also run the code!
  - ↪ Can we get Stateful Incremental JIT Compilation?

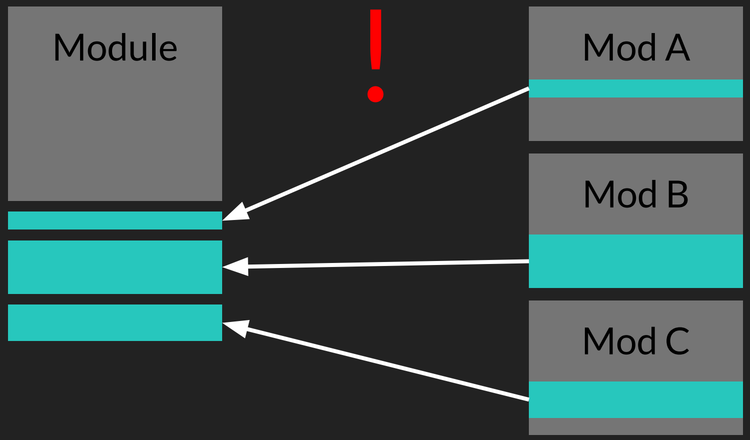
Stateful Incremental JIT Compilation

ThinLTO Summaries help

- Requires even more housekeeping
  - Track file changes on disk
    Source Module Hashes
  - Determine source dependencies for functions
    Call Graph
  - Decide what needs to be recompiled
  - Link against existing code
    Name Promotion
  - Rename duplicate globals and resolve to existing ones
  - Invalidate existing function handles (and rebind?)

Name Promotion in ThinLTO Backends

→ Names of imported globals must not clash with local names!



Name Promotion in ThinLTO Backends

main.cpp

```
int readGlobal();
void writeGlobal(int x);

static int GlobalVar = 10;

int main() {
  writeGlobal(1);
  return readGlobal() + GlobalVar;
}
```

```
$ clang main.cpp global.cpp -o test
$ ./test
$ echo $?
11
```

Name collision on import

global.cpp

```
static int GlobalVar = 0;
int readGlobal() { return GlobalVar; }
void writeGlobal(int x) { GlobalVar = x; }
```

Name Promotion in ThinLTO Backends

Imported pseudo-code main.cpp

```
int readGlobal() { return GlobalVar.llvm.98887675656456543548795; }
void writeGlobal(int x) { GlobalVar.llvm.98887675656456543548795 = x; }

static int GlobalVar = 10;
extern int GlobalVar.llvm.98887675656456543548795;

int main() {
  writeGlobal(1);
  return readGlobal() + GlobalVar;
}
```

Postfix with source module hash

→ Like the regular linker, JIT can use hash to find module with the symbol definition

Stateful Incremental Builds

global.cpp

```cpp
#include <cstdio>

static int GlobalVar = 0;
int readGlobal() { return GlobalVar; }

void writeGlobal(int x) {
  printf("Modified GlobalVar");
  GlobalVar = x;
}
```

→ Modification changes module hash

Stateful Incremental Builds

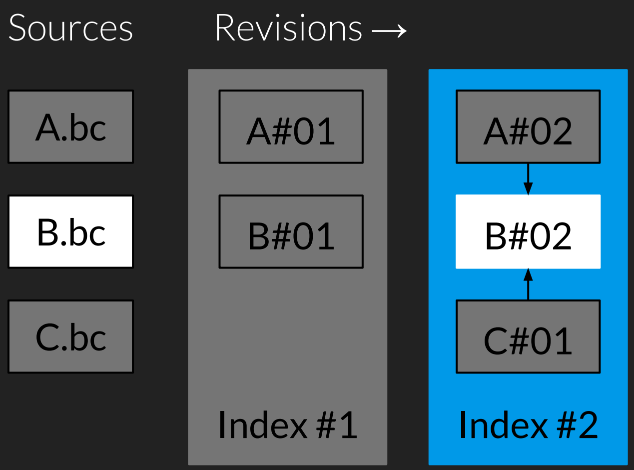
Adjusted pseudo-code global.cpp

```
#include <cstdio>

extern int GlobalVar.llvm.988876756564565435487 95;
int readGlobal() { return GlobalVar.llvm.988876756564565435487 95; }

void writeGlobal(int x) {
  printf("Modified GlobalVar");
  GlobalVar.llvm.988876756564565435487 95 = x;
}
```

→ Manually rename own global, so it can be resolved to existing instance in previous revision

Stateful Incremental Builds

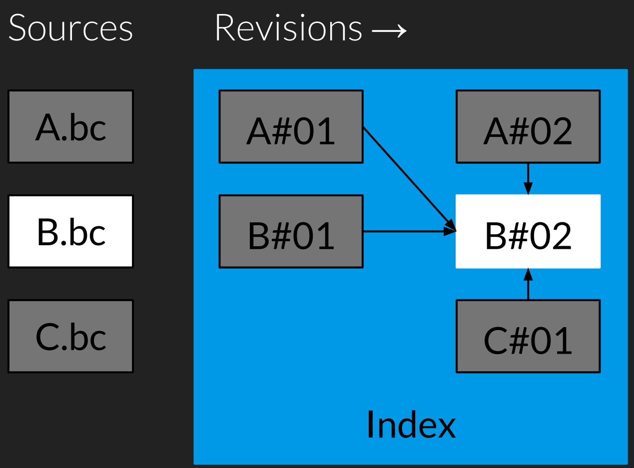
→ ThinLTO tools tailored for ahead-of-time compilation, e.g.
`ModuleSummaryIndex` uses module ID as key and not Hash



- Cannot (easily) keep one Index over multiple revisions.
- Diff summaries to obtain on-disk changes during Thin-Link
- Promote stateful globals manually in Thin Backends (lots of extra maintenance!)

Stateful Incremental Builds

→ If `ModuleSummaryIndex` keys were hashes, we could easily import from previous revisions:



Stateful Incremental Builds

- Other ThinLTO implementations used:
  - `llvm::getLazyBitcodeModule()`
  - `llvm::readModuleSummaryIndex()`
  - `llvm::computeDeadSymbols()`
  - `llvm::ComputeCrossModuleImportForModule()`

Overlap: ThinLTO ↔ (Stateful) Incremental JIT

- Require global view of the program
- Computationally expensive & huge scale
- Run in two stages
  - Global analysis (sequential)
  - Per-module recompilation (in parallel)
- Focus on optimizations in the compiled program
  - JIT: keep acceptable runtime with high granularity!

Short-Term Goals

- Hack Clang to emit Hashes in `ModuleSummaryIndex`.
- Build infrastructure upon ThinLTO implementations that directly targets stateful incremental recompilation to get rid of hacky workarounds.
- Extend management infrastructure, e.g. handle cases like function signature changes.
- Load ninja.build descriptions instead of single files.

Long-Term Ideas

- Git-like revision management for program state
  - Fast-forward merge side-effect-free changes
  - Let developer resolve conflicts
  - Allow saving and restoring state snapshots
  - Fall back to restart on fail

Long-Term Ideas

- Git-like revision management for program state
- Emit program summary data to executable (or extra file)
  - Like debug information for program state

Long-Term Ideas

- Git-like revision management for program state
- Emit program summary data to executable (or extra file)
  - Like debug information for program state
- Package JIT as dylib, so it can be used like a sanitizer

Long-Term Ideas

- Git-like revision management for program state
- Emit program summary data to executable (or extra file)
  - Like debug information for program state
- Package JIT as dylib, so it can be used like a sanitizer
- Implement support in common developer tools (e.g. LLDB) to modify executables based on program summary data

Thanks!

Let’s dive into the code!

https://github.com/weliveindetail/ThinLtoJit