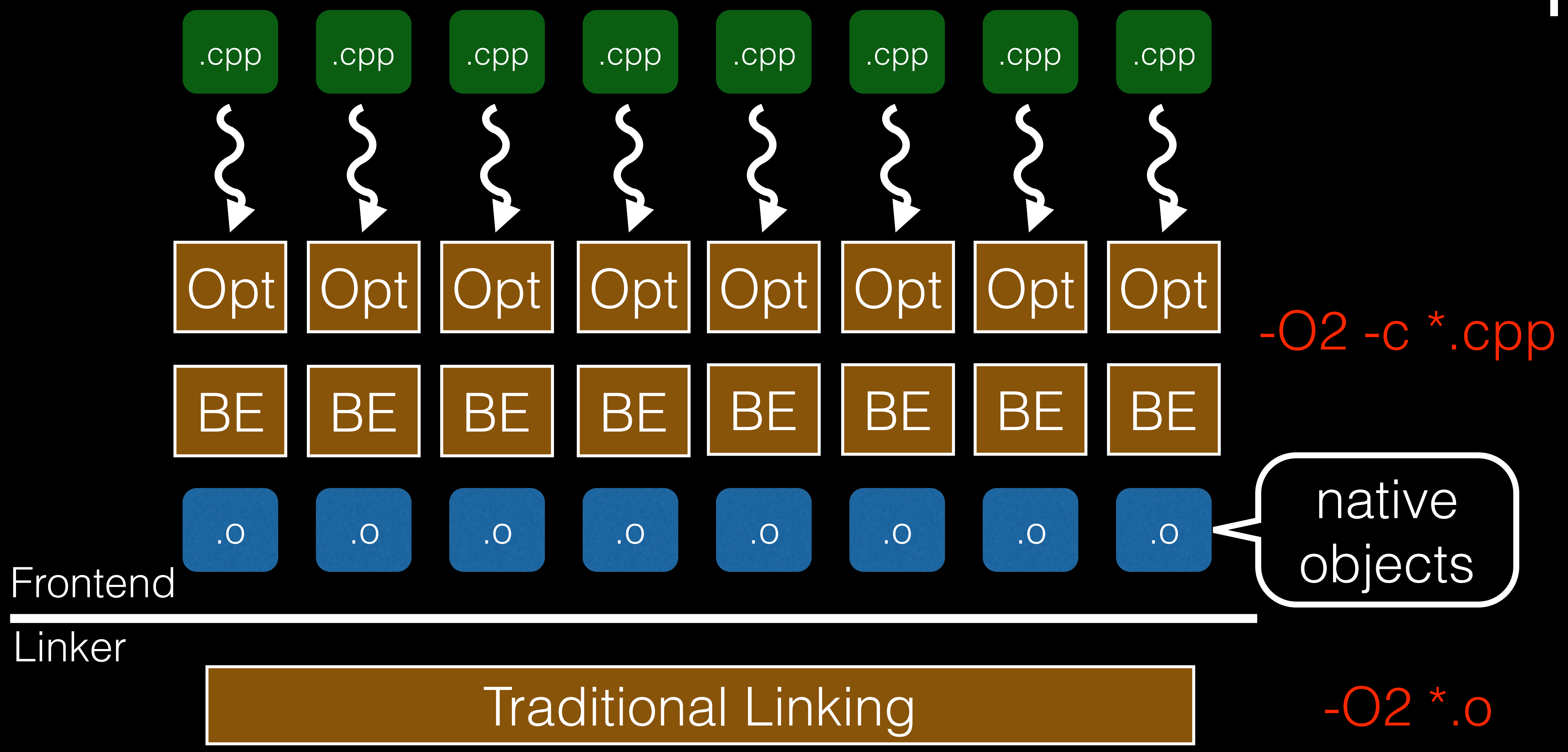
ThinLTO

Towards Always-Enabled LTO

Scalable and Incremental Link-Time Optimization

Teresa Johnson   Mehdi Amini   Xinliang David Li

Traditional Compilation

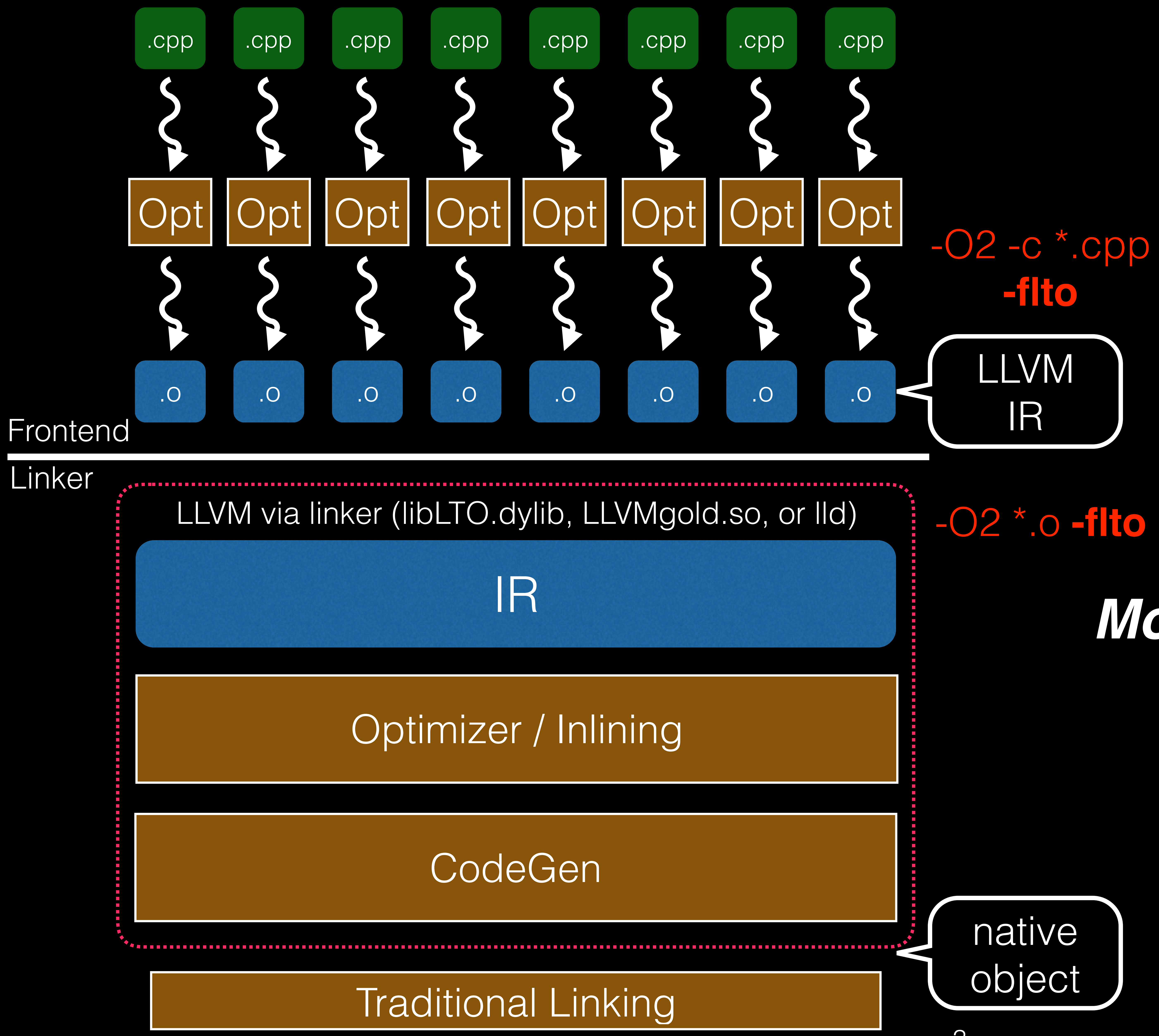


Highly parallel frontend processing

IR Optimizations

CodeGen

LLVM LTO: in a Nutshell



Highly parallel frontend processing

+ initial optimizations

Link all bitcode in one single Module

***Monolithic** LTO Implementation*

IR Optimizations

CodeGen

Why LTO?

Benefits

+ Performance! 10% boost is common.
  - ➔ Removes module optimization boundaries via Cross-Module Optimization (CMO)
  - ➔ Most of benefit comes from cross-module inlining

+ Binary size: inherent dead-stripping and auto-hidden visibility *via* internalization.

Performance Improvements - Execution Time	Δ	Previous	Current	σ
SingleSource/Benchmarks/Shootout-C++/objinst	-96.80%	0.0937	0.0030	-
SingleSource/Benchmarks/Dhrystone/fldry	-93.17%	0.0893	0.0061	0.0002
SingleSource/Benchmarks/Dhrystone/dry	-79.21%	0.0534	0.0111	0.0000
SingleSource/Benchmarks/Misc/matmul_f64_4x4	-52.76%	0.0199	0.0094	0.0008
SingleSource/UnitTests/Vector/build2	-46.56%	0.0421	0.0225	0.0002
MultiSource/Benchmarks/tramp3d-v4/tramp3d-v4	-35.45%	0.1839	0.1187	0.0026
SingleSource/Benchmarks/Misc/flops-6	-33.50%	0.0403	0.0268	0.0001
MultiSource/Benchmarks/Olden/bh/bh	-31.09%	0.1412	0.0973	0.0002
MultiSource/Benchmarks/Olden/power/power	-30.45%	0.0867	0.0603	0.0000

Single source improvements because global variables can be internalized (better alias analysis, etc.).

➔ LTO is more powerful than “Unity Build” because of **Linker supplied information.**

Example

main.c 〉 No Selection

```c
#include <stdio.h>

// Defined in a.c
int foo1(void);


void foo4(void) {
  printf("Hi\n");
}


int main() {
  return foo1();
}
```

a.c 〉 No Selection

```c
void foo4(void); // Defined in main.c

static signed int i = 0;

void foo2(void) {
  i = -1;
}
static int foo3() {
  foo4();
  return 10;
}
int foo1(void) {
  int data = 0;
  if (i < 0)
    data = foo3();
  data = data + 42;
  return data;
}
```

lto.cpp 〉 No Selection

```c
static signed int i = 0;
void foo2(void) {
  i = -1;
}
static int foo3() {
  foo4();
  return 10;
}
int foo1(void) {
  int data = 0;
  if (i < 0)
    data = foo3();
  data = data + 42;
  return data;
}
void foo4(void) {
  printf("Hi\n");
}
int main() {
  return foo1();
}
```

Optimization across *module boundaries!*

+

Linker Information

Example

main.c 〉 No Selection

```c
#include <stdio.h>

// Defined in a.c
int foo1(void);


void foo4(void) {
  printf("Hi\n");
}


int main() {
  return foo1();
}
```

a.c 〉 No Selection

```c
void foo4(void); // Defined in main.c

static signed int i = 0;

void foo2(void) {
  i = -1;
}
static int foo3() {
  foo4();
  return 10;
}
int foo1(void) {
  int data = 0;
  if (i < 0)
    data = foo3();
  data = data + 42;
  return data;
}
```

lto.cpp 〉 No Selection

```c
static signed int i = 0;



static int foo3() {
  foo4();
  return 10;
}
static int foo1(void) {
  int data = 0;
  if (i < 0)
    data = foo3();
  data = data + 42;
  return data;
}
static void foo4(void) {
  printf("Hi\n");
}
int main() {
  return foo1();
}
```

Optimization across *module boundaries!*

+

Linker Information

Example

main.c 〉 No Selection

```c
#include <stdio.h>

// Defined in a.c
int foo1(void);


void foo4(void) {
  printf("Hi\n");
}


int main() {
  return foo1();
}
```

a.c 〉 No Selection

```c
void foo4(void); // Defined in main.c

static signed int i = 0;

void foo2(void) {
  i = -1;
}
static int foo3() {
  foo4();
  return 10;
}
int foo1(void) {
  int data = 0;
  if (i < 0)
    data = foo3();
  data = data + 42;
  return data;
}
```

lto.cpp 〉 No Selection

```c
static int foo3() {
  foo4();
  return 10;
}
static int foo1(void) {
  int data = 0;


  data = data + 42;
  return data;
}



int main() {
  return 42;
}
```

Optimization across *module boundaries!*

+

Linker Information

Example

main.c 〉 No Selection

```c
#include <stdio.h>

// Defined in a.c
int foo1(void);


void foo4(void) {
  printf("Hi\n");
}


int main() {
  return foo1();
}
```

a.c 〉 No Selection

```c
void foo4(void); // Defined in main.c

static signed int i = 0;

void foo2(void) {
  i = -1;
}
static int foo3() {
  foo4();
  return 10;
}
int foo1(void) {
  int data = 0;
  if (i < 0)
    data = foo3();
  data = data + 42;
  return data;
}
```

lto.cpp 〉 No Selection

```cpp
int main() {
  return 42;
}
```

Optimization across *module boundaries!*

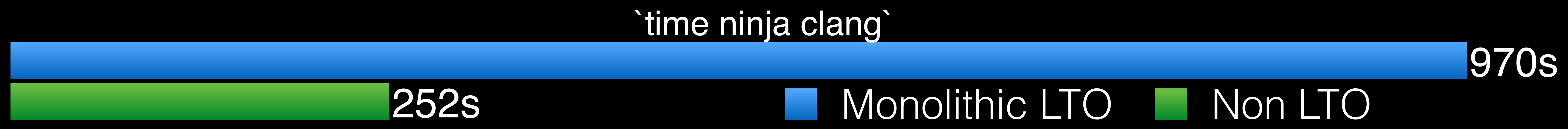
+

Linker Information

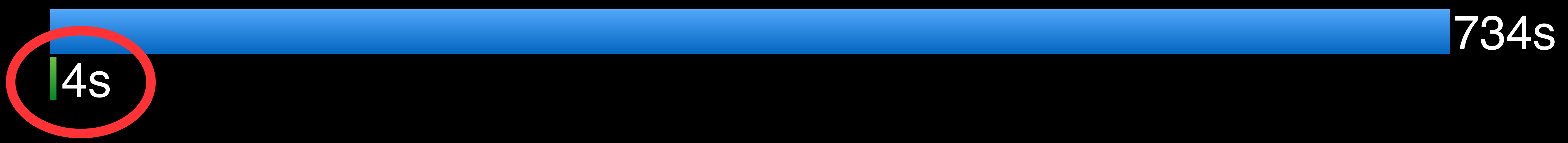
Monolithic LTO: No Free Lunch!

LTO adoption is still low after >10 years of existence: why?

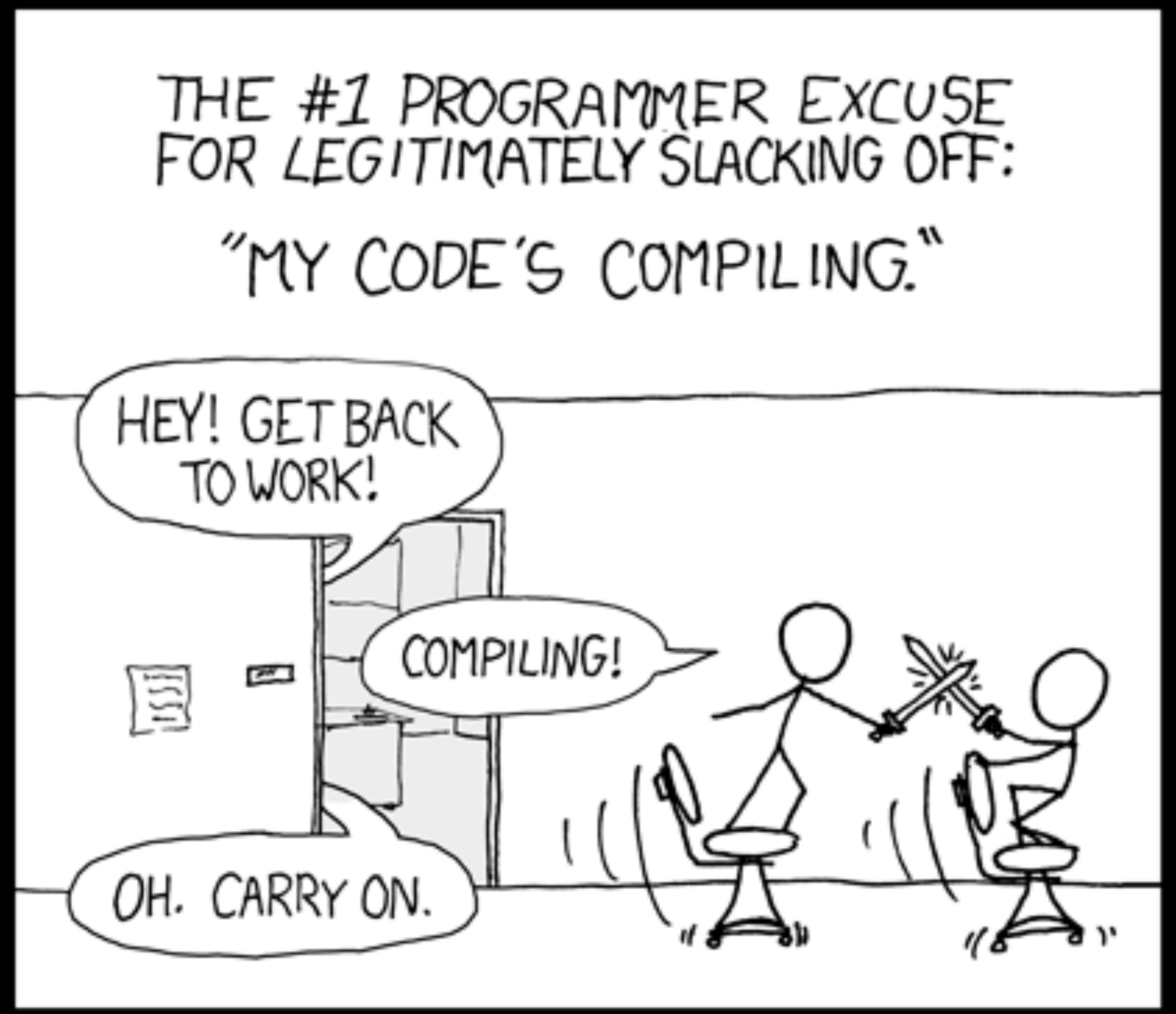
+ **Slow**: inherently serial / can't be distributed



+ Not friendly with **incremental build**: fix a typo, and see the full program being re-optimized as a whole.

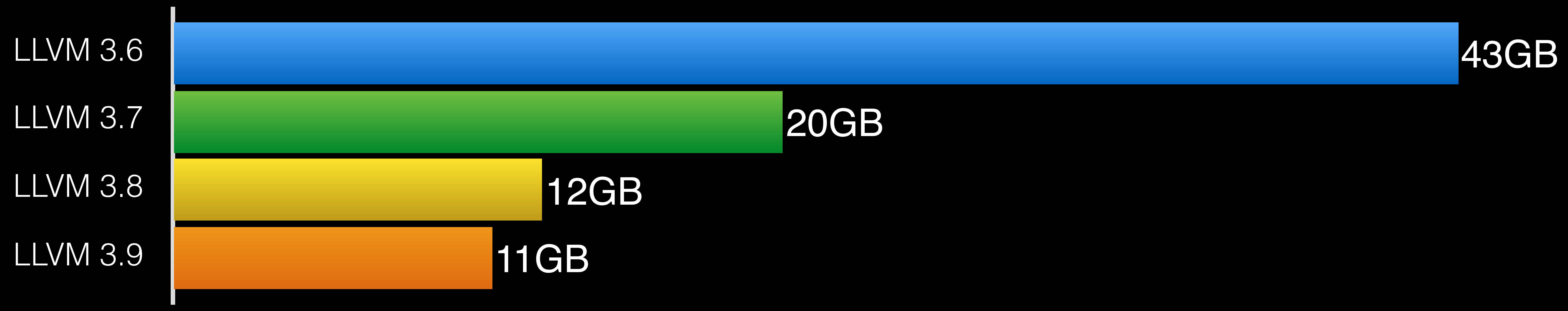


+ **Memory hungry**: all the program in memory.



https://www.xkcd.com/303/

Malloc Peak for the LTO Link of the `clang-3.6` binary, with only the X86 backend configured



Dead end: we killed the link of Chromium with debug info after >3h and >50GB mem

Monolithic LTO: No Free Lunch!

LTO adoption is still low after >10 years of existence: why?

+ **Slow**: inherently serial / can't be distributed

→ Parallel

+ Not friendly with **incremental build**: fix a typo, and see the full program being re-optimized as a whole.

→ Incremental

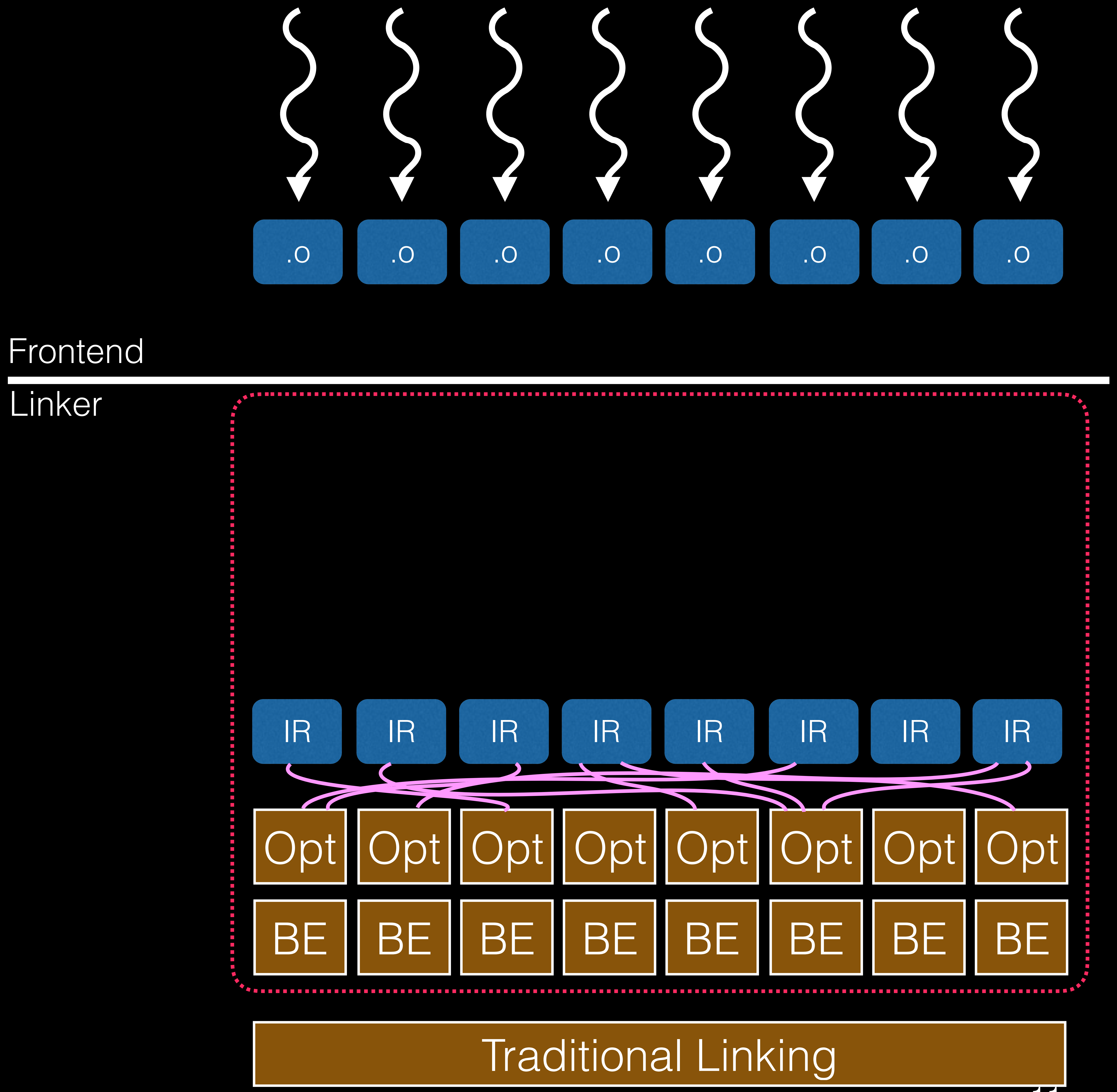
+ **Memory hungry**: all the program in memory.

→ Memory lean

To fulfill this goal of LTO always enabled,

we need a solution designed around these three key features!

ThinLTO: Design



- **Parallel**

  Fully parallel compile step and backends, enabling distributed builds

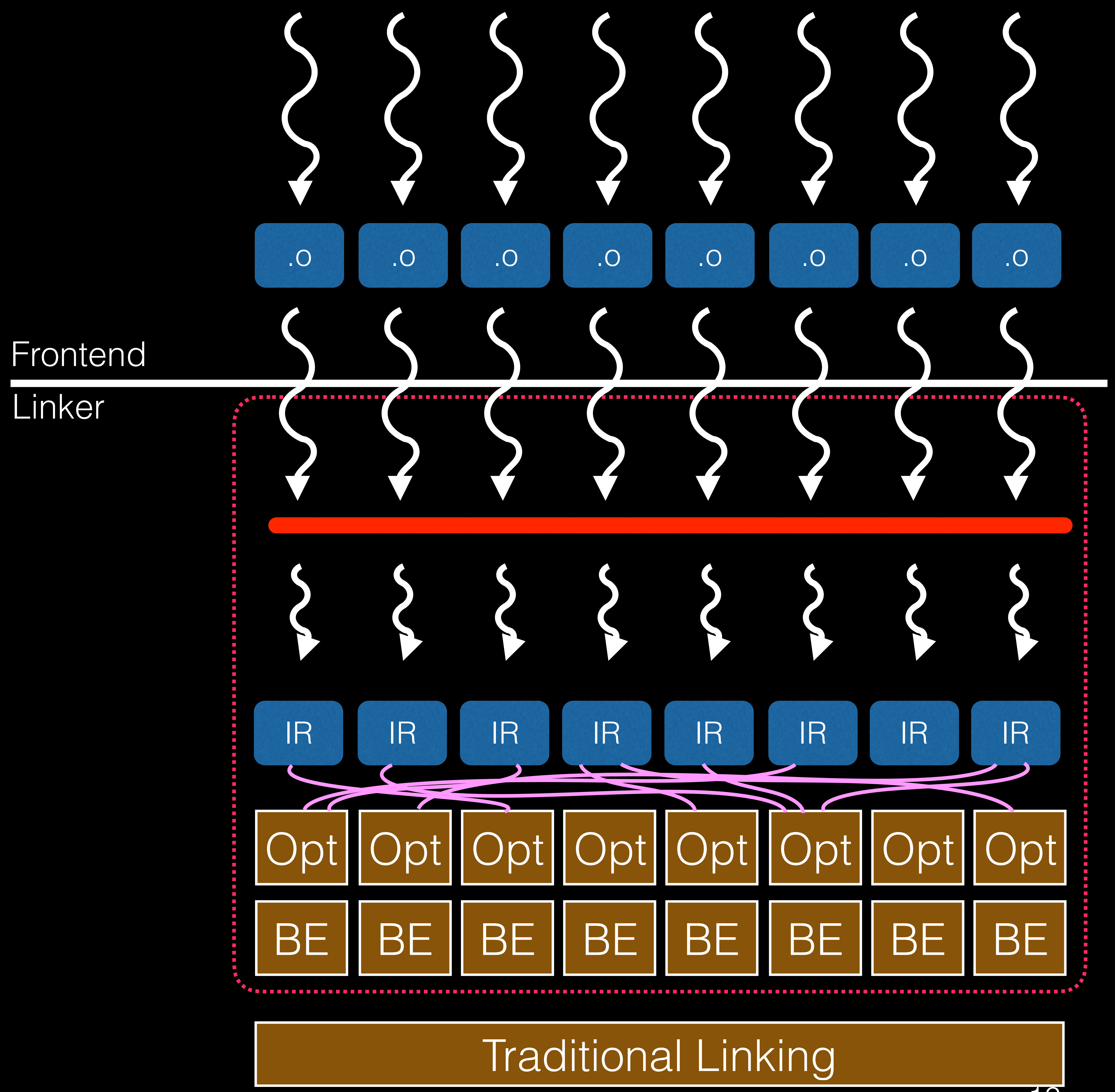
- **Incremental**

  Module is unit of compilation

- **Memory lean**

  Only perform profitable cross module optimization into each module

ThinLTO: Design



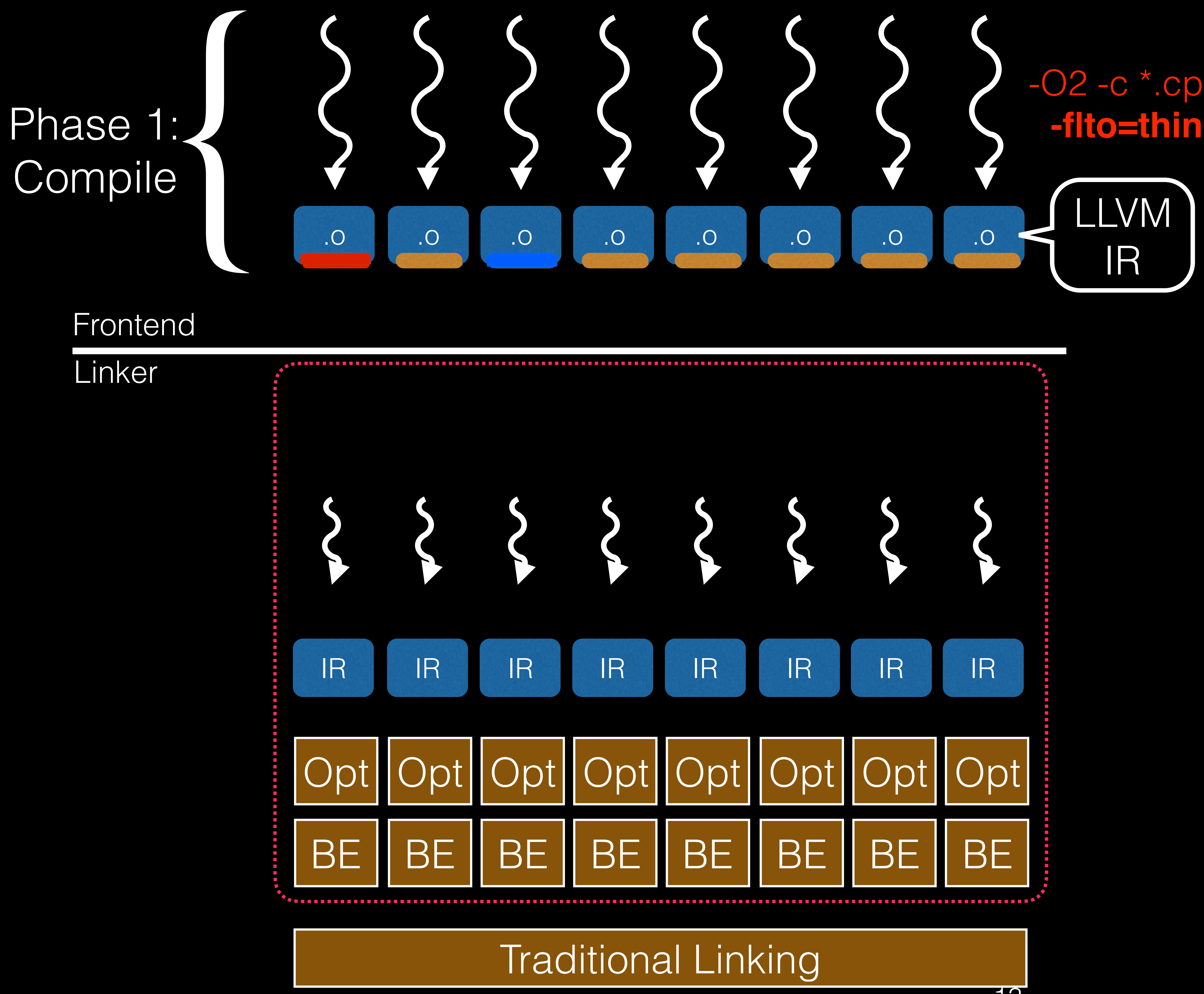
→ Parallel

→ Incremental

***Thin** serial synchronization step*

→ Memory lean

ThinLTO: Design



Fully-parallel frontend processing
+ initial optimizations

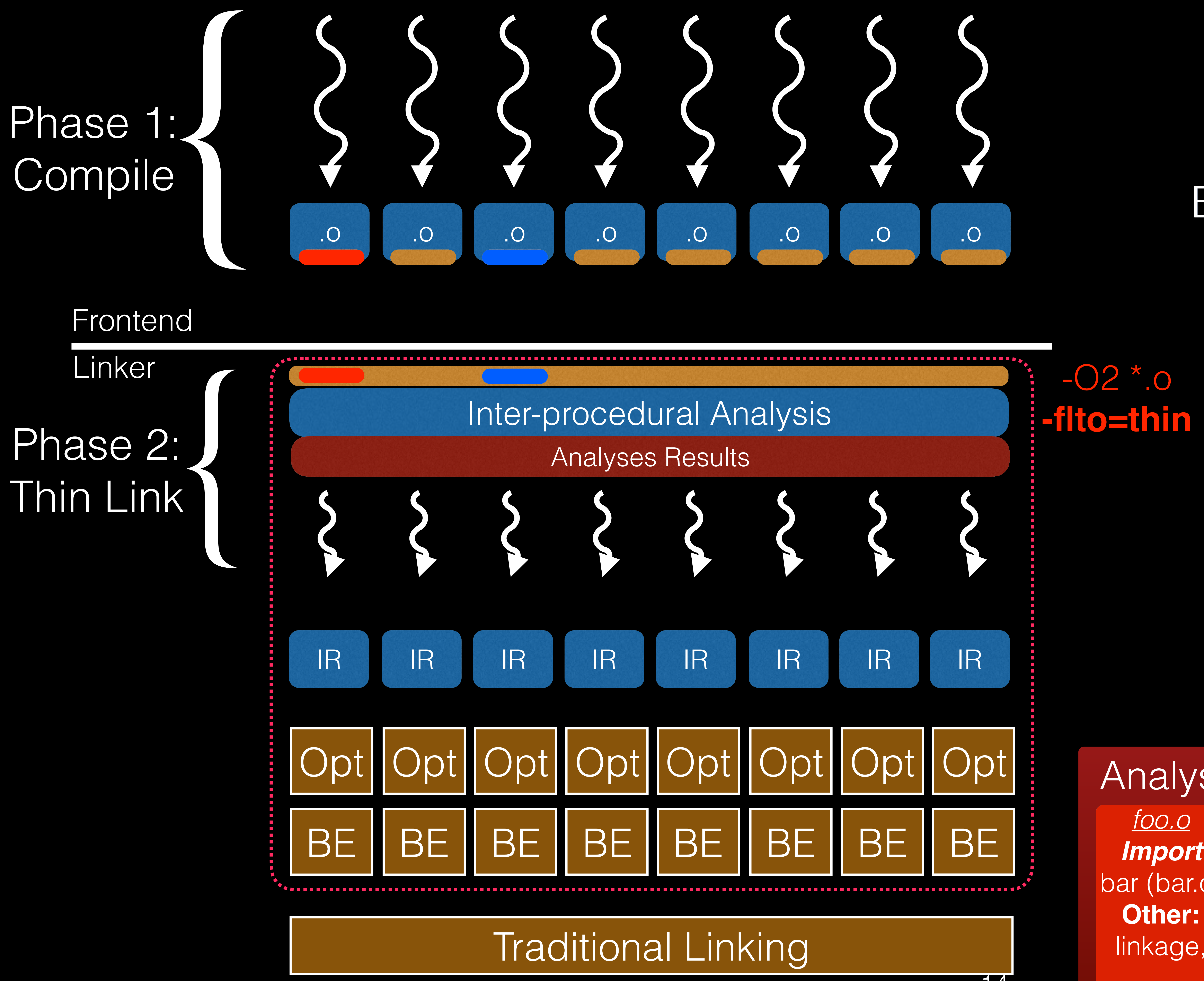
Extra per-function summary information
are generated *"on the side"*

- Includes local reference/call graph
- Includes module hash
  (for incremental build)

foo: 20 insts, *call bar* (foo.o)

bar: 10 insts (bar.o)

ThinLTO: Design



Fully-parallel frontend processing
+ initial optimizations

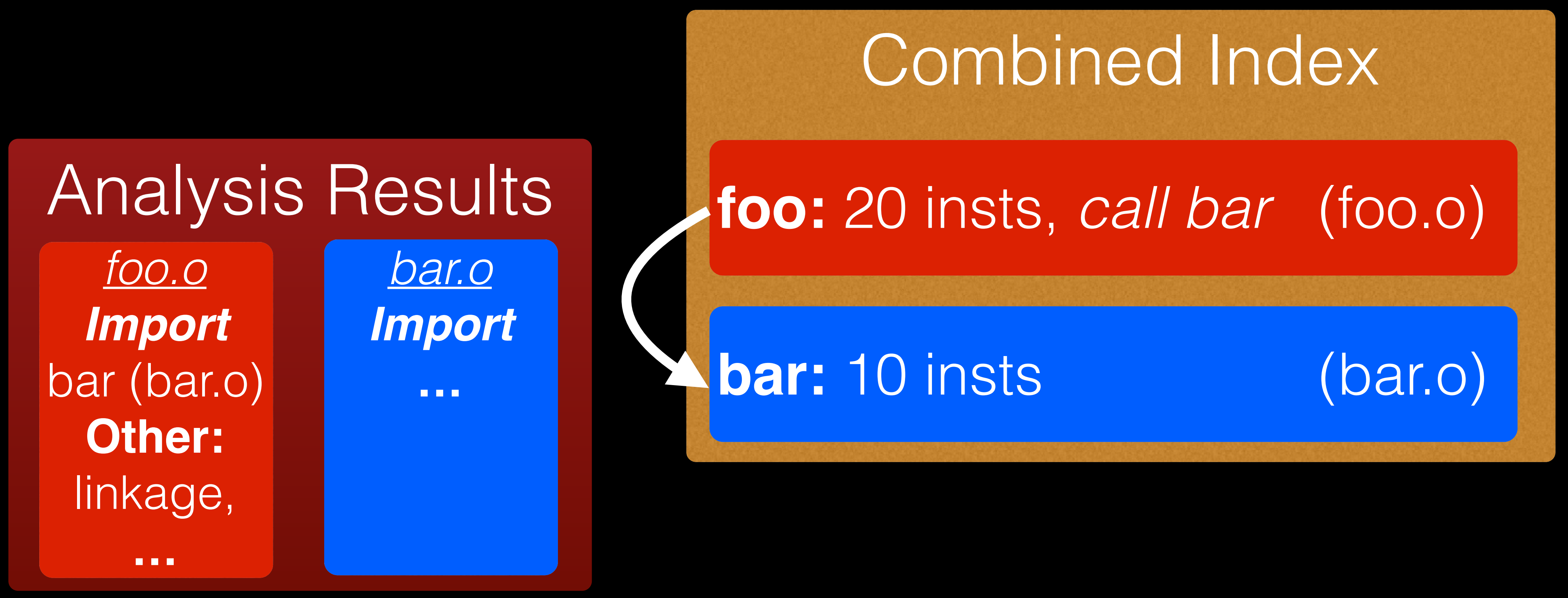
Extra per-function summary information are generated *"on the side"*

Link only the summary info in a giant index: *thin-link*.

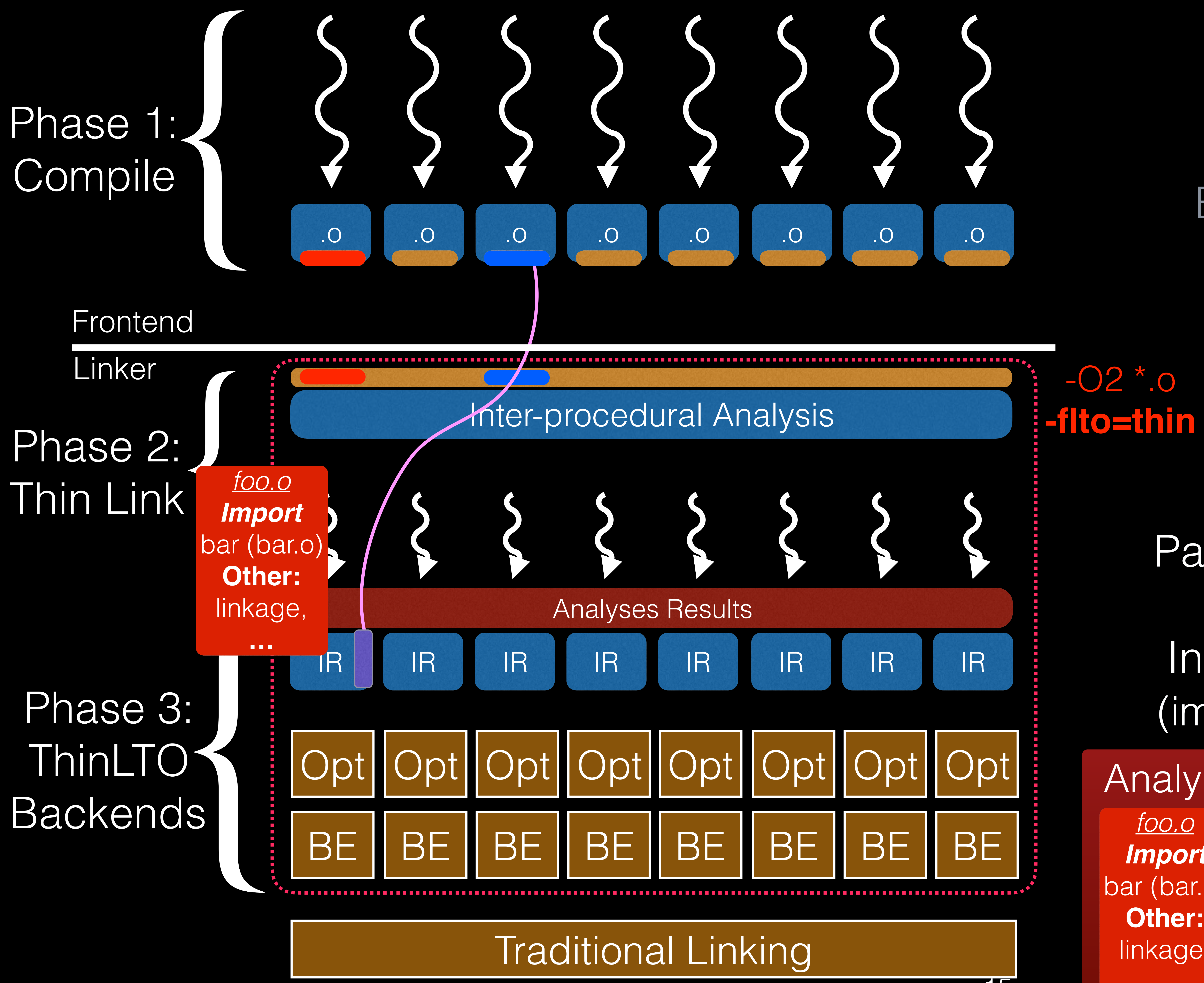
- Includes full reference/call graph to enable Inter-procedural Analysis (IPA)

No need to parse the IR

Serial phase - but very fast!



ThinLTO: Design

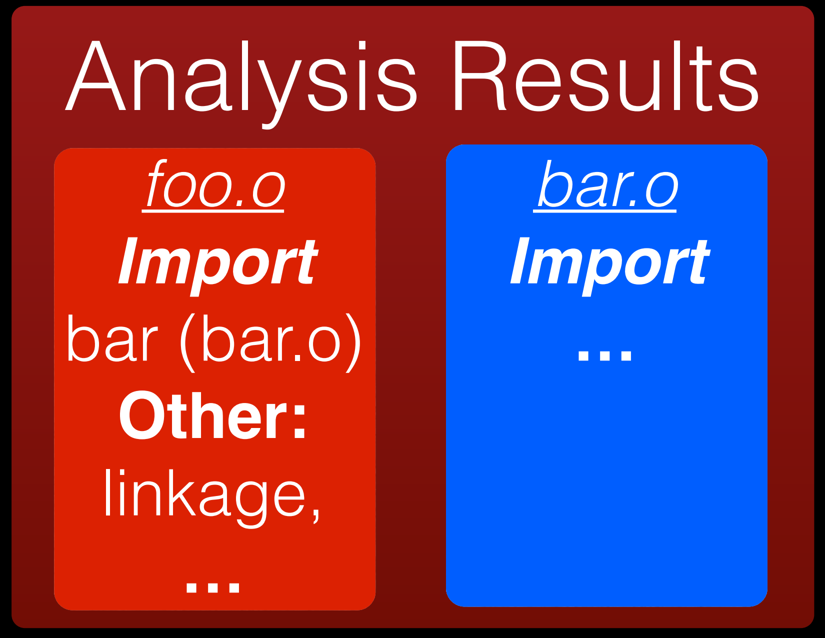


Fully-parallel frontend processing
+ initial optimizations

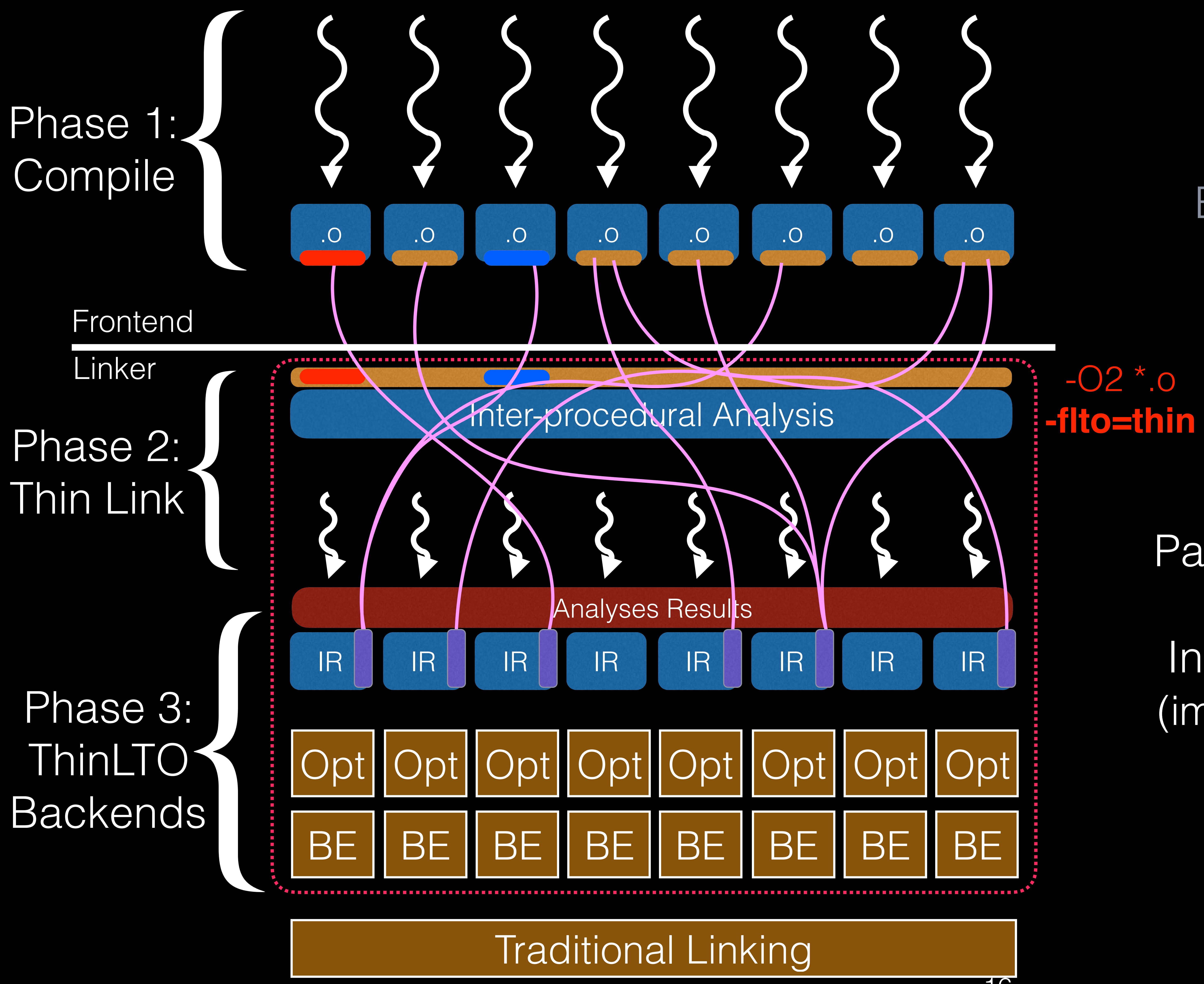
Extra per-function summary information are generated *"on the side"*

Link only the summary info in a giant index: *thin-link*.
- Includes full reference/call graph to enable Inter-procedural Analysis (IPA)

Parallel Inter-procedural transformations based on the analyses results
Includes cross-module function importing (imported functions are dropped after inlining)



ThinLTO: Design



Fully-parallel frontend processing
+ initial optimizations

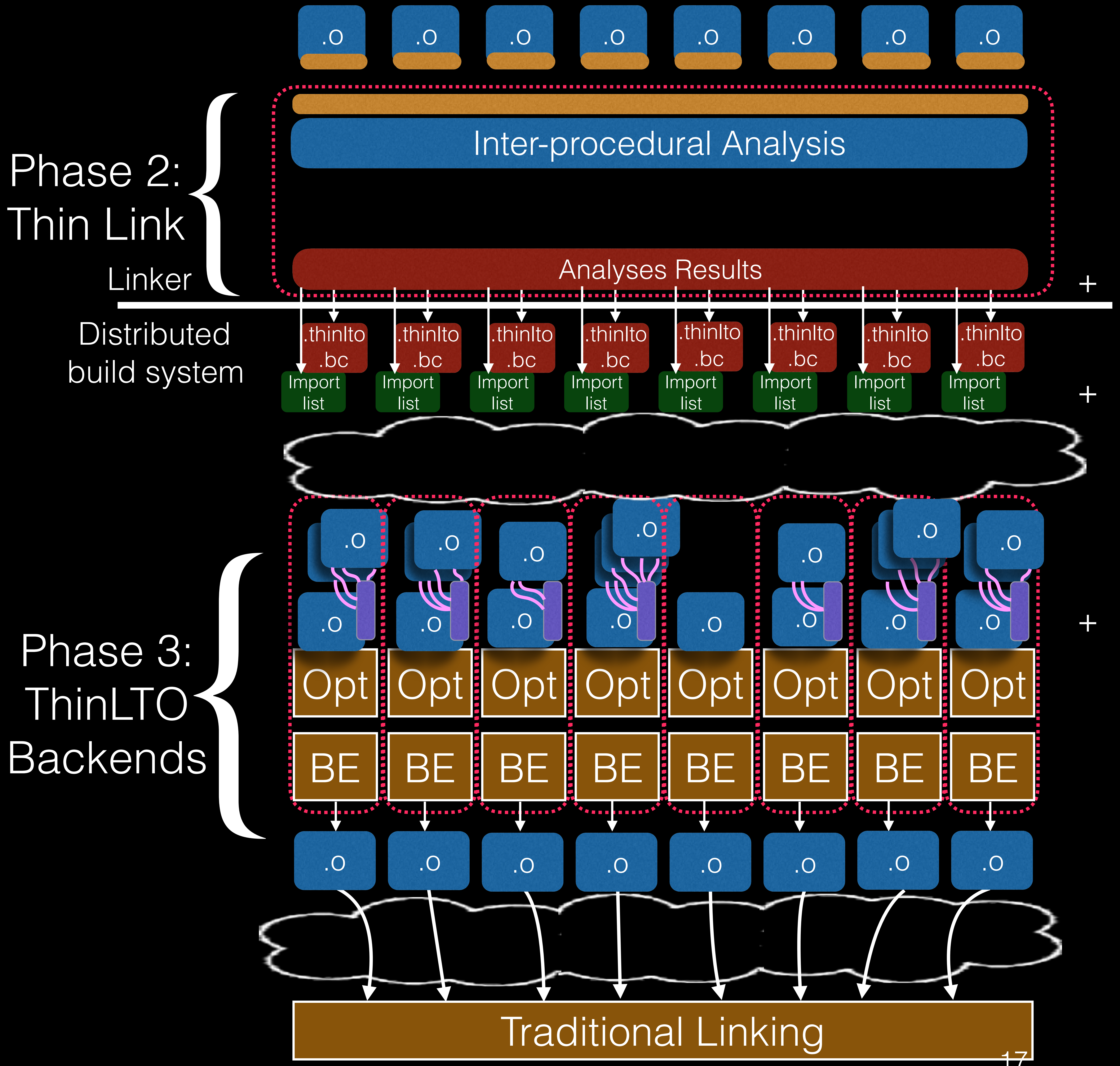
Extra per-function summary information
are generated *"on the side"*

Link only the summary info in a
giant index: *thin-link*.
- Includes full reference/call graph to
enable Inter-procedural Analysis (IPA)

Parallel Inter-procedural transformations
based on the analyses results
Includes cross-module function importing
(imported functions are dropped after inlining)

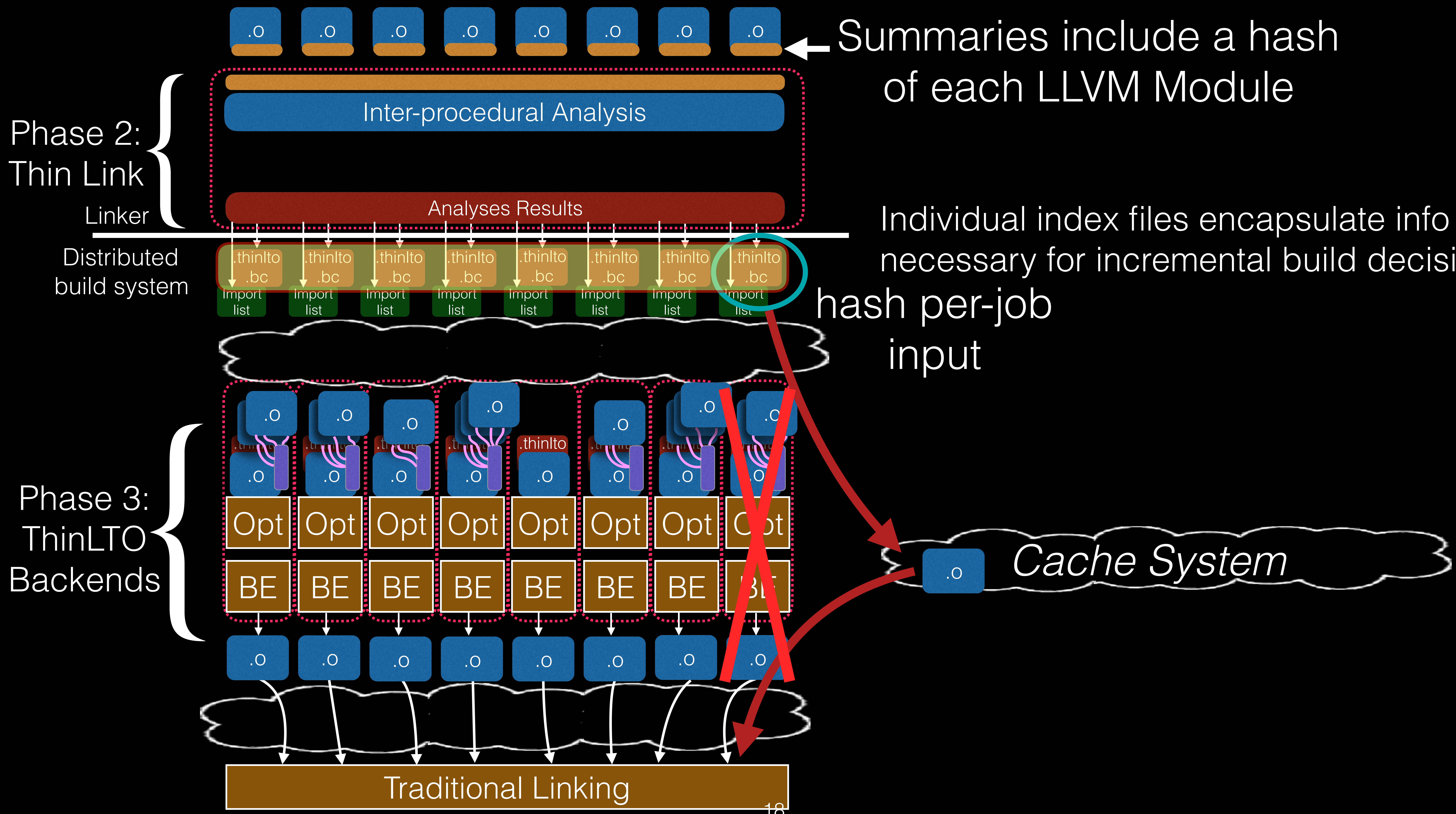
Fully-parallel (very boring) usual
optimizations and CodeGen

ThinLTO: Distributed Builds



+ Stream out analysis results for each backend job
+ Produce dependent module lists: the build system bundles these for each job
+ ***Distributed*** Inter-procedural transformations based on the analyses results

ThinLTO: Incremental Builds

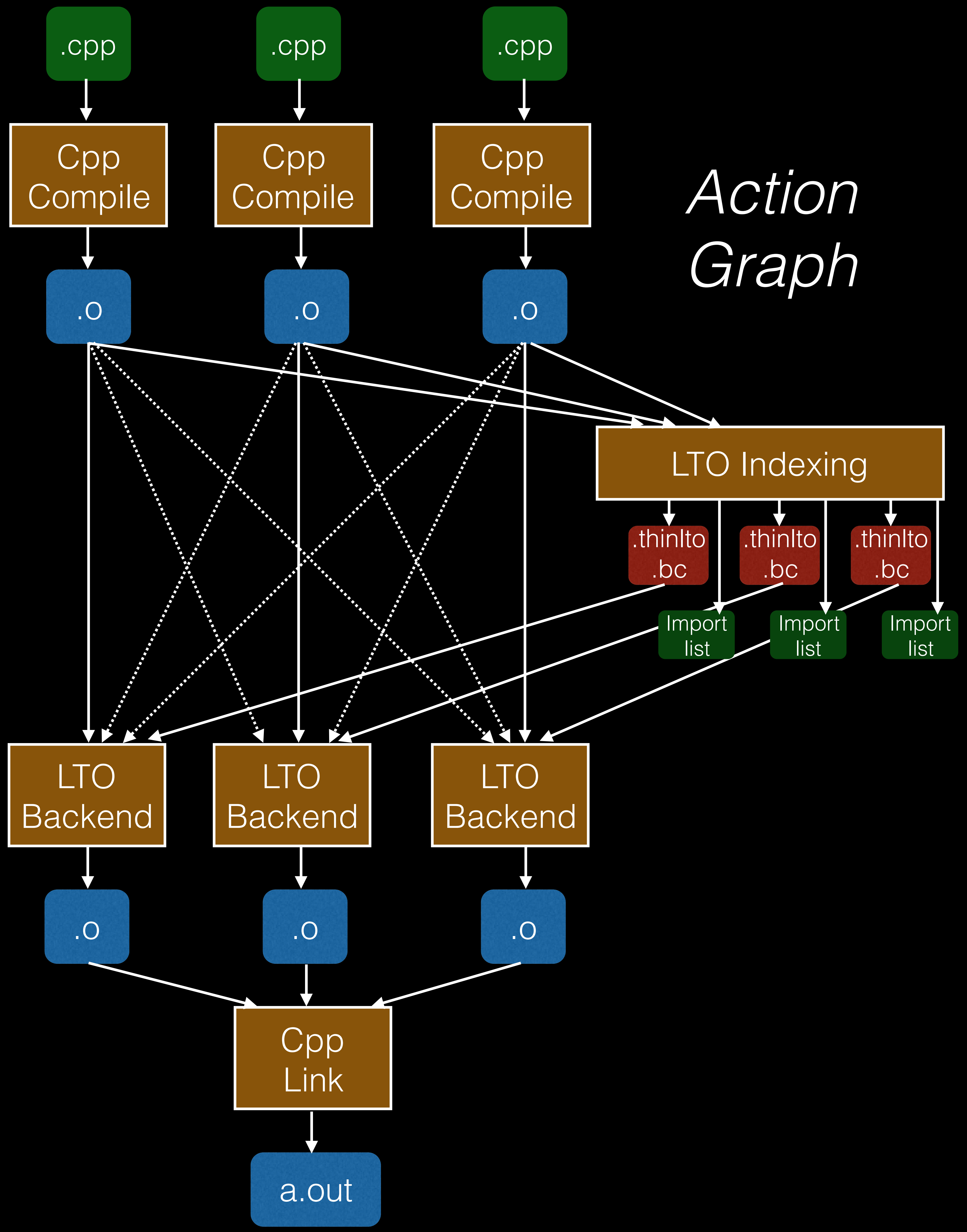


Summaries include a hash of each LLVM Module

Individual index files encapsulate info necessary for incremental build decisions

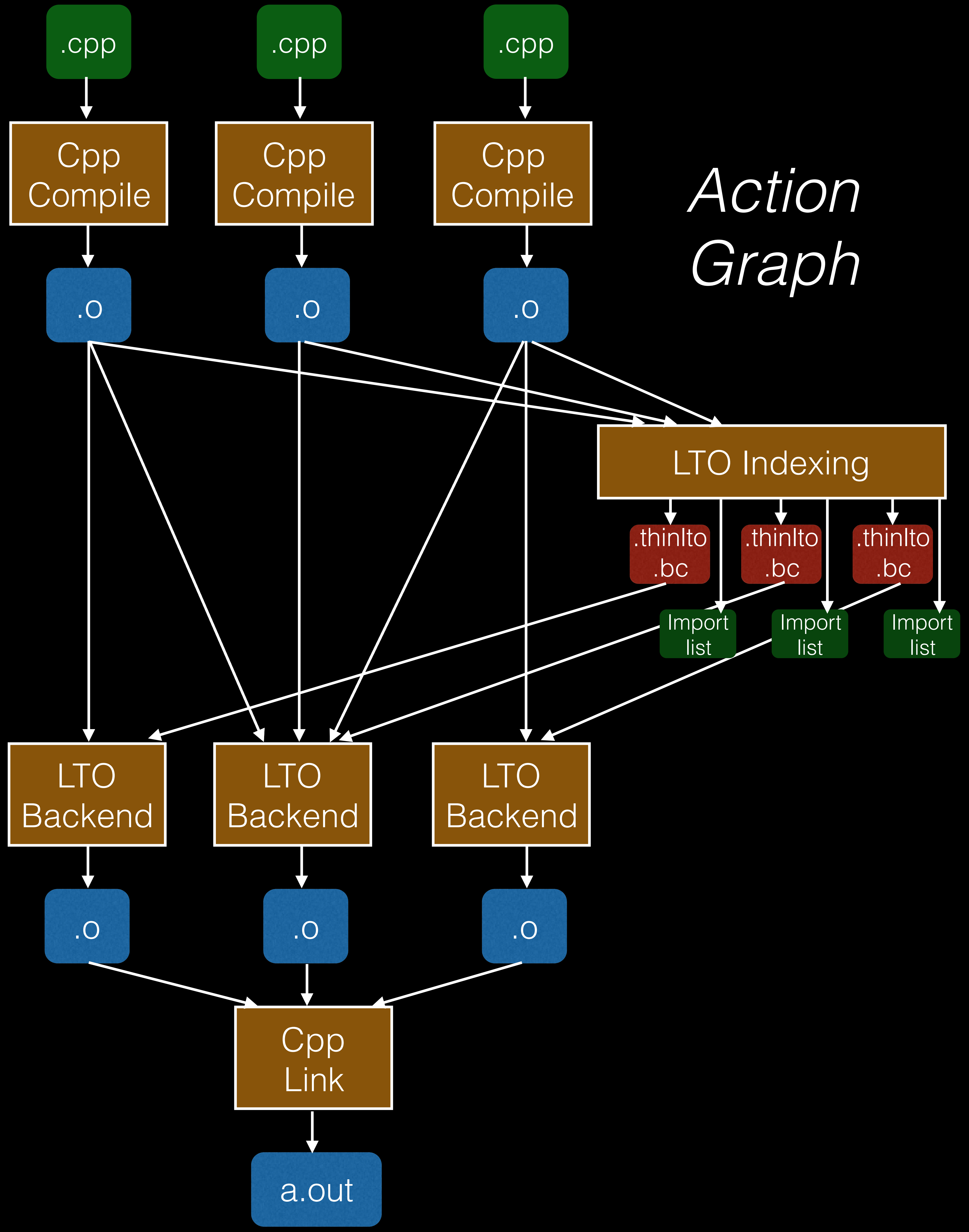
hash per-job input

Integration with Bazel Build System



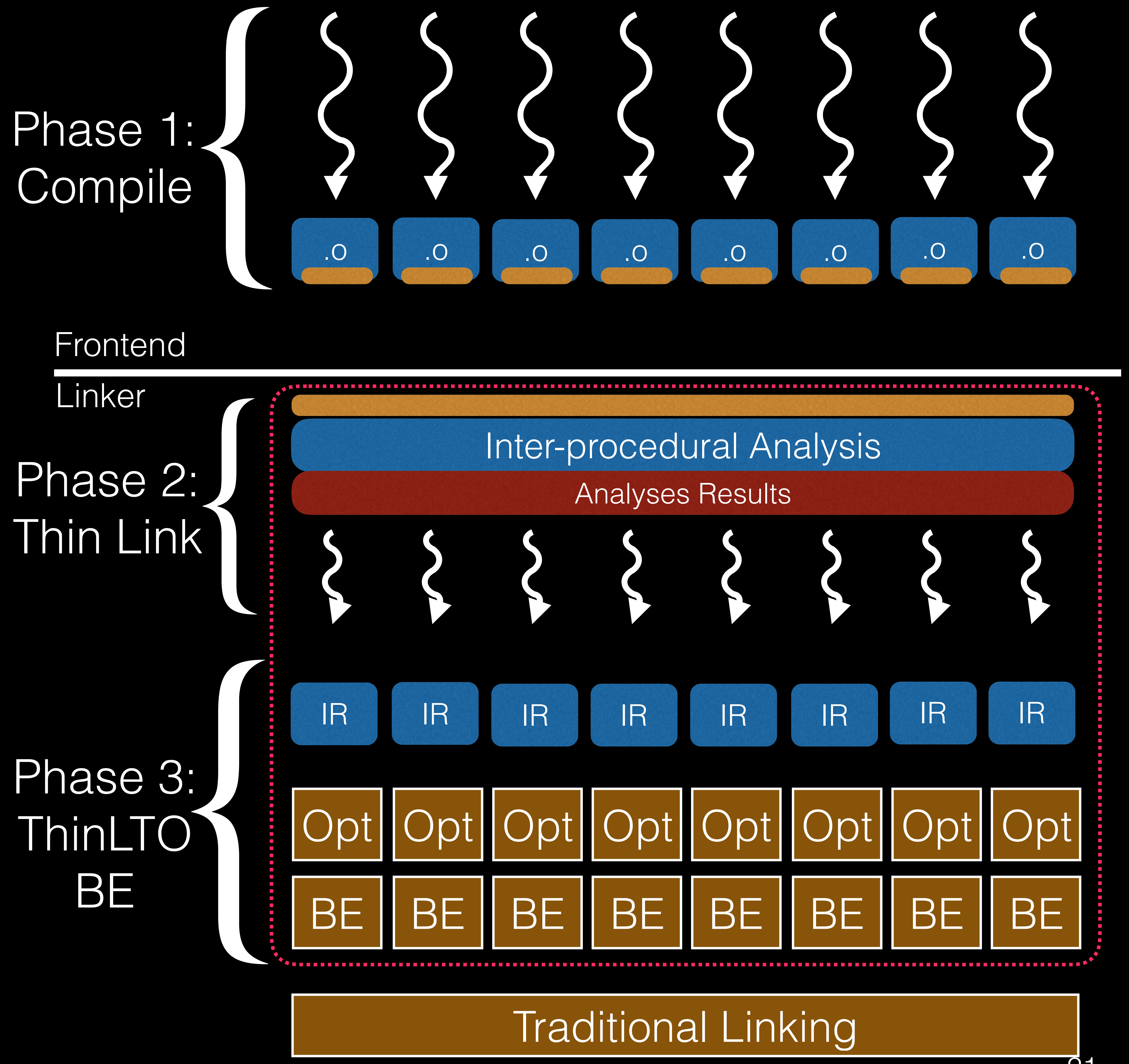
- Open source Google build system
  - ➡ https://bazel.build/
- Action Graph connecting inputs to outputs via actions
  - ➡ Constructed from dependency graph
- Build walks the action graph
  - ➡ Each action can be sent to a different remote build node

Integration with Bazel Build System



- Open source Google build system
  - ➡ https://bazel.build/
- Action Graph connecting inputs to outputs via actions
  - ➡ Constructed from dependency graph
- Build walks the action graph
  - ➡ Each action can be sent to a different remote build node
- Import lists are used to finalize the inputs to the LTO Backend actions
- Caching is content-based (hash of inputs)

ThinLTO Whole Program Optimization



Cross-module optimization split into two parts:

1. Analysis during Thin Link
   + Operates on full call/reference graph created from summaries
   + Results recorded in the index (e.g. linkage type changes)

2. Transformation during Parallel Backends
   + Applies results of whole program analysis performed during Thin Link
   + Independently applied in each backend

ThinLTO WPA: Compile Time Optimization
Weak Linkage Resolution

vector1.cpp 〉 No Selection

```cpp
#include <vector>

void foo(std::vector<int> &V) {
  V.push_back(1);
  V.push_back(2);
  V.push_back(3);
  V.push_back(4);
}
```

vector2.cpp 〉 No Selection

```cpp
#include <vector>

void bar(std::vector<int> &V) {
  V.push_back(11);
  V.push_back(22);
  V.push_back(33);
  V.push_back(44);
}
```

```llvm
source_filename = "vector2.cpp"
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.12.0"

define void @_Z3fooRNSt3__16vectorIiNS_9allocatorIiEEEE(%"class.std::__1::vector"* dereferenceable(24)) #0 {
    ;   Lot
    ;   of
    ; stuff
  ret void
}

define linkonce_odr void @_ZNSt3__16vectorIiNS_9allocatorIiEEE21__push_back_slow_pathIiEEvOT_(%"class.std::__1::vector"*,
        i32* nocapture readonly dereferenceable(4)) #0 align 2 personality i8* bitcast (i32 (...)* @__gxx_personality_v0 to i8*) {
  %3 = getelementptr inbounds %"class.std::__1::vector", %"class.std::__1::vector"* %0, i64 0, i32 0, i32 1
  %4 = bitcast i32** %3 to i64*
  %5 = load i64, i64* %4, align 8, !tbaa !6
  %6 = bitcast %"class.std::__1::vector"* %0 to i64*
  %7 = load i64, i64* %6, align 8, !tbaa !11
  %8 = sub i64 %5, %7
  %9 = ashr exact i64 %8, 2
  %10 = add nsw i64 %9, 1
  %11 = icmp ugt i64 %10, 4611686018427387903
  br i1 %11, label %12, label %15

; <label>:12                                      ; preds = %2
```

… ~100 instructions

C++ template generates a lot of redundant code!

+ Regular O2 must codegen copies in both objects, the linker picks one
+ Monolithic LTO will *merge* these and codegen only one naturally
+ ThinLTO selects one at Thin Link time ➔ other copies marked in index for drop after inlining

Linking clang: **~25%** less functions are codegen!

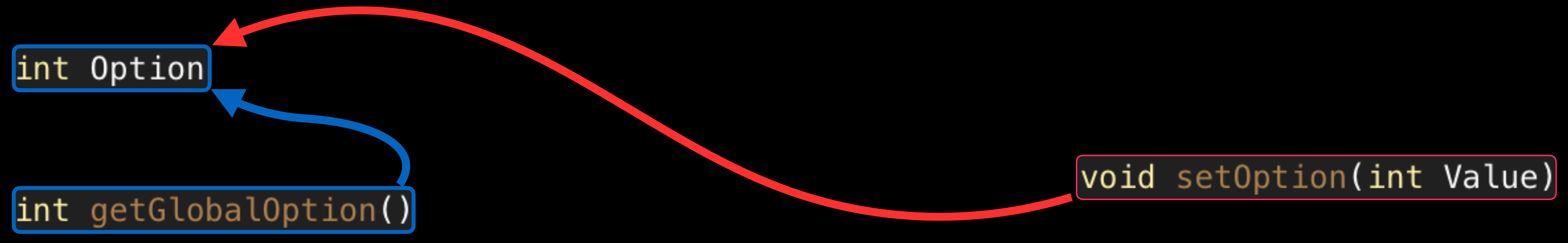
ThinLTO WPA: Dead Global Pruning

```
int Option = 42;

int getGlobalOption() {
  return Option;
}
```

```
extern int Option;

void setOption(int Value) {
  Option = Value;
}
```



- Linker identifies external reference to getGlobalOption()
- Compute reachability to externally referenced nodes in index
- Prune unreachable nodes from the graph

Linker Info:
external ref.

getGlobalOption

➔ **Enabler** for better subsequent analyses
  - *Option* can be *internalized* and later constant folded.
  - The function-importing will generate a smaller list ➔ save CPU cycles!

Profile Guided Optimization (PGO)

```
void foo() {
  if (usuallyTrue)
    hot();
  else
    cold();
}

void hot() {
  // more code
}
void cold {
  // very small
}
```

```
void foo() {
  if (usuallyTrue)
    // more code
  else
    cold();
}

void cold() {
  // very small
}
```

+ More likely to inline small cold function without profile data
+ With profile data, call hotness is known

➔ Inline hot calls more aggressively for better optimization of hot path

Profile Guided Optimization (PGO)

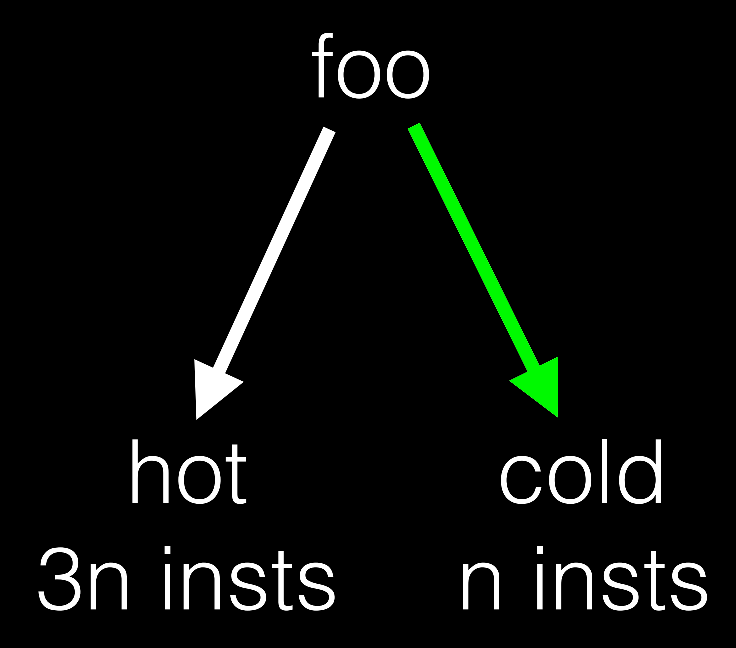
foo.cc

```
void foo() {
  if (usuallyTrue)
    hot();
  else
    cold();
}
```

bar.cc

```
void hot() {
  // more code
}
void cold {
  // very small
}
```

Thin Link Call Graph



+ More likely to import small cold function without profile data

Profile Guided Optimization (PGO)

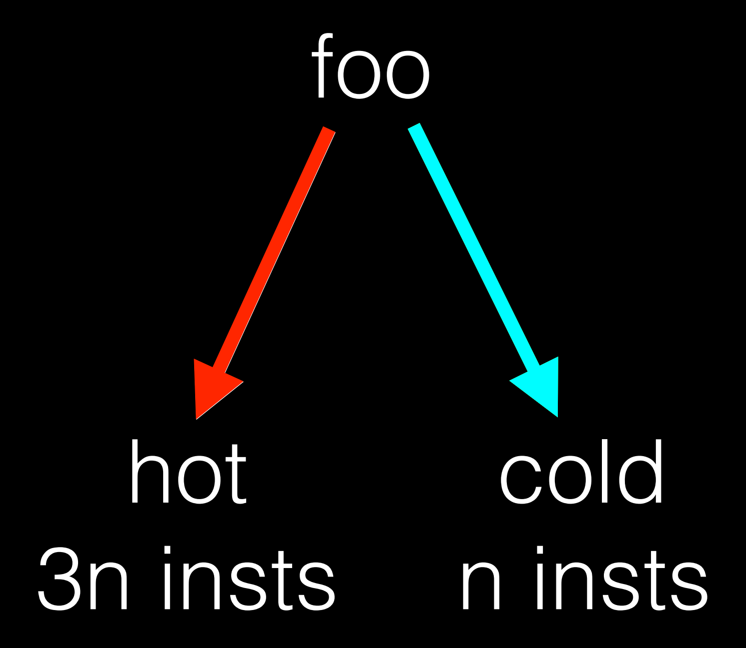
foo.cc

```
void foo() {
   if (usuallyTrue)
      hot();
   else
      cold();
}
```

bar.cc

```
void hot() {
   // more code
}
void cold {
   // very small
}
```

Thin Link Call Graph



- More likely to import small cold function without profile data
- With profile data, call hotness is known
  - ➔ Annotate edges in thin link graph with hotness
  - ➔ Import hot calls more aggressively to match inlining heuristics

- Not unusual for 98% of functions to be cold
  - ➔ Reduce compile time by avoiding needless importing

Profile Guided Optimization (PGO): Indirect Call Promotion

indirect_call.cpp 〉 No Selection

```cpp
extern int (*bar)();

int bar_impl() {
  // do stuff
  // ...
  return value;
}

int foo() {
  // do stuff
  // ...
  if (bar == &bar_impl)
    return bar_impl()
  return bar(); // value profile: bar_impl
}
```

Indirect Call Promotion can only promote if the target is in the same module.

PGO Indirect Call Promotion

indirect_target.cpp 〉 No Selection

```cpp
int bar_impl() {
  // do stuff
  // ...
  return value;
}

int (*bar)() = &bar_impl;


```

indirect_call.cpp 〉 No Selection

```cpp
extern int (*bar)();

int foo() {
  // do stuff
  // ...


  return bar(); // value profile: bar_impl
}
```

Summary:
  foo: calls bar_impl

The summary records hot indirect call targets as regular calls (speculative).

→ Hot indirect call targets imported, available for promotion & inlining

Thin Link Call Graph

foo → bar_impl

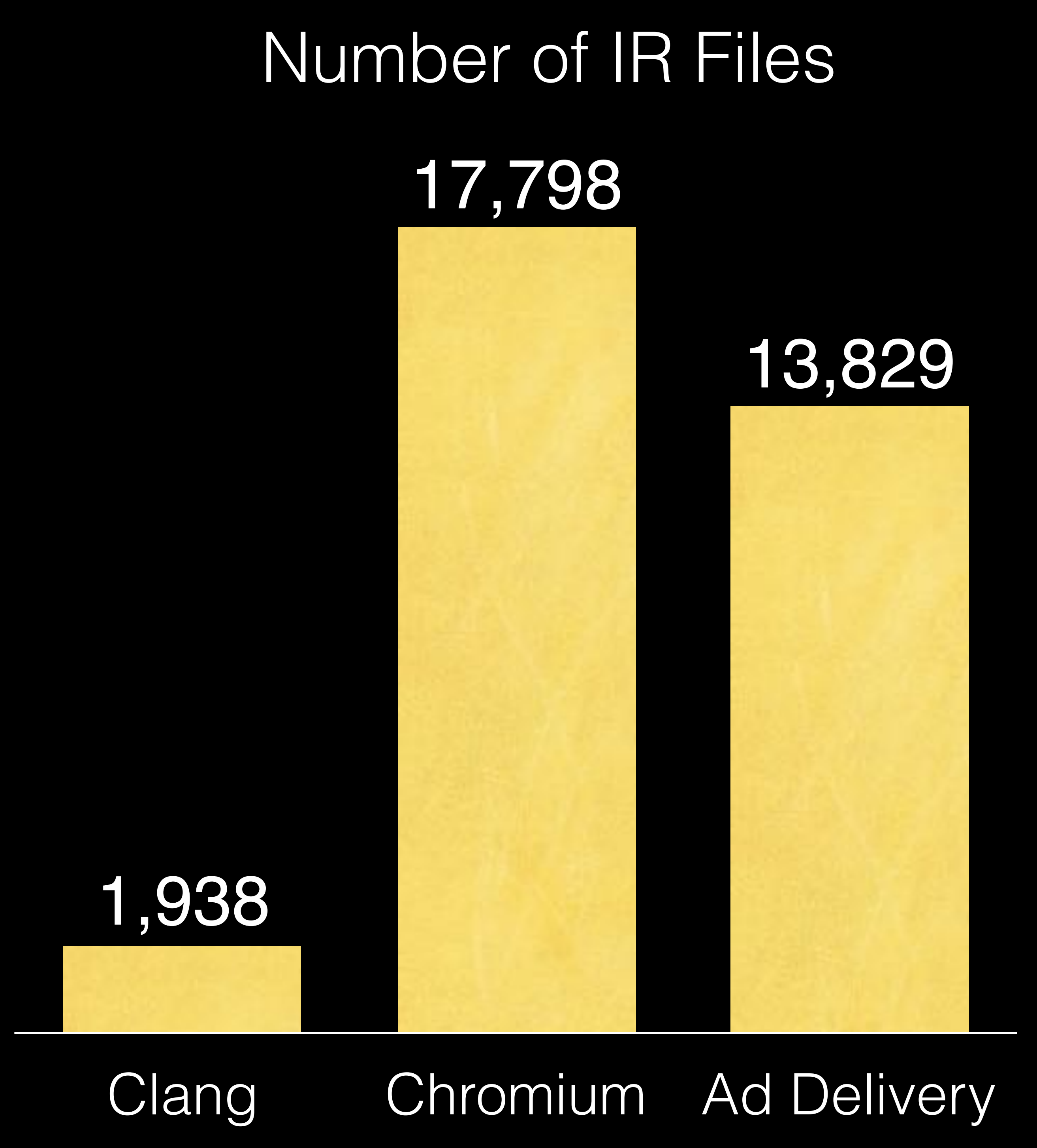
Build Time Comparison with GCC LTO

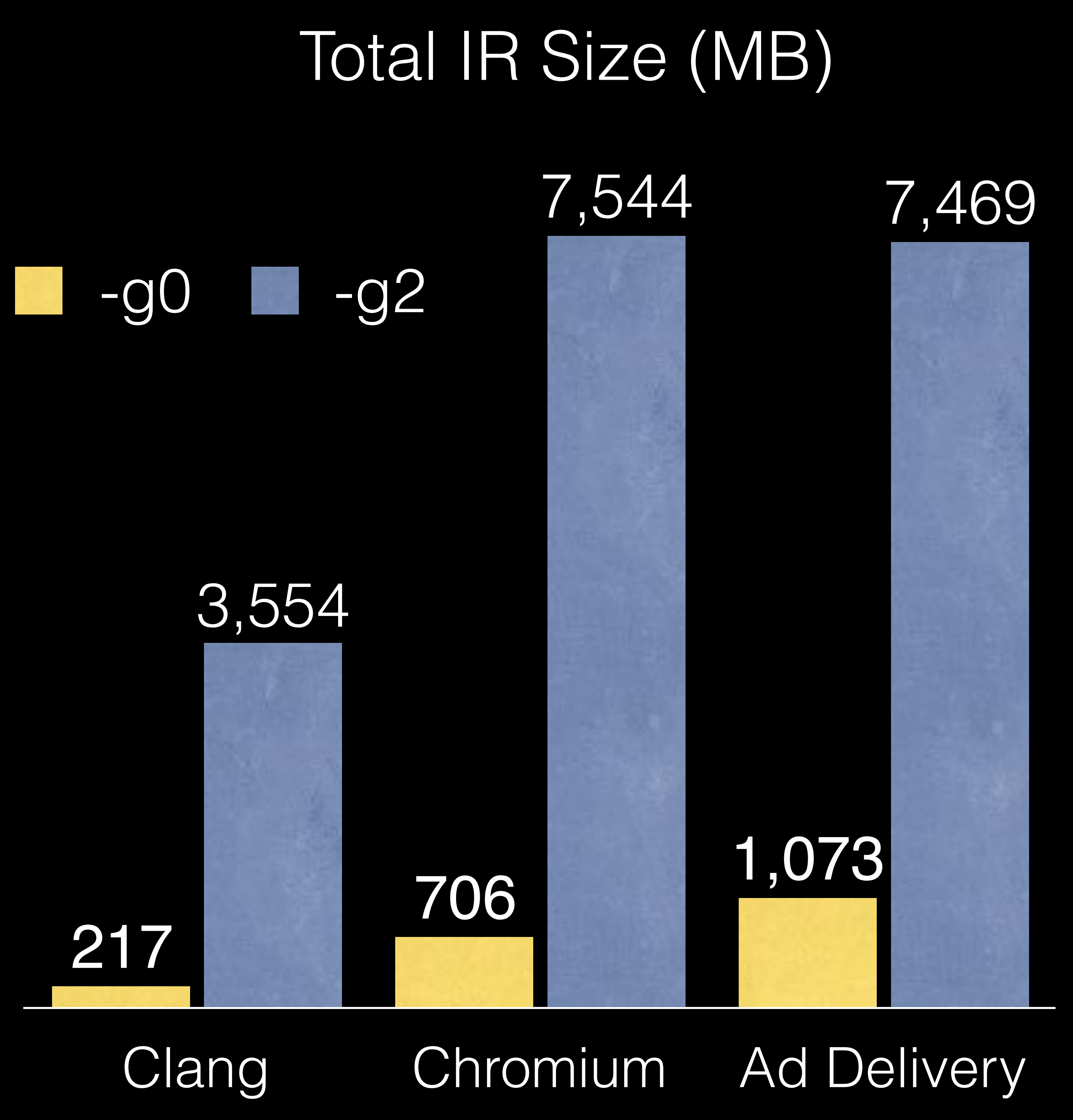
GCC has a mature and well-tuned sophisticated LTO implementation (WHOPR), with two parts:

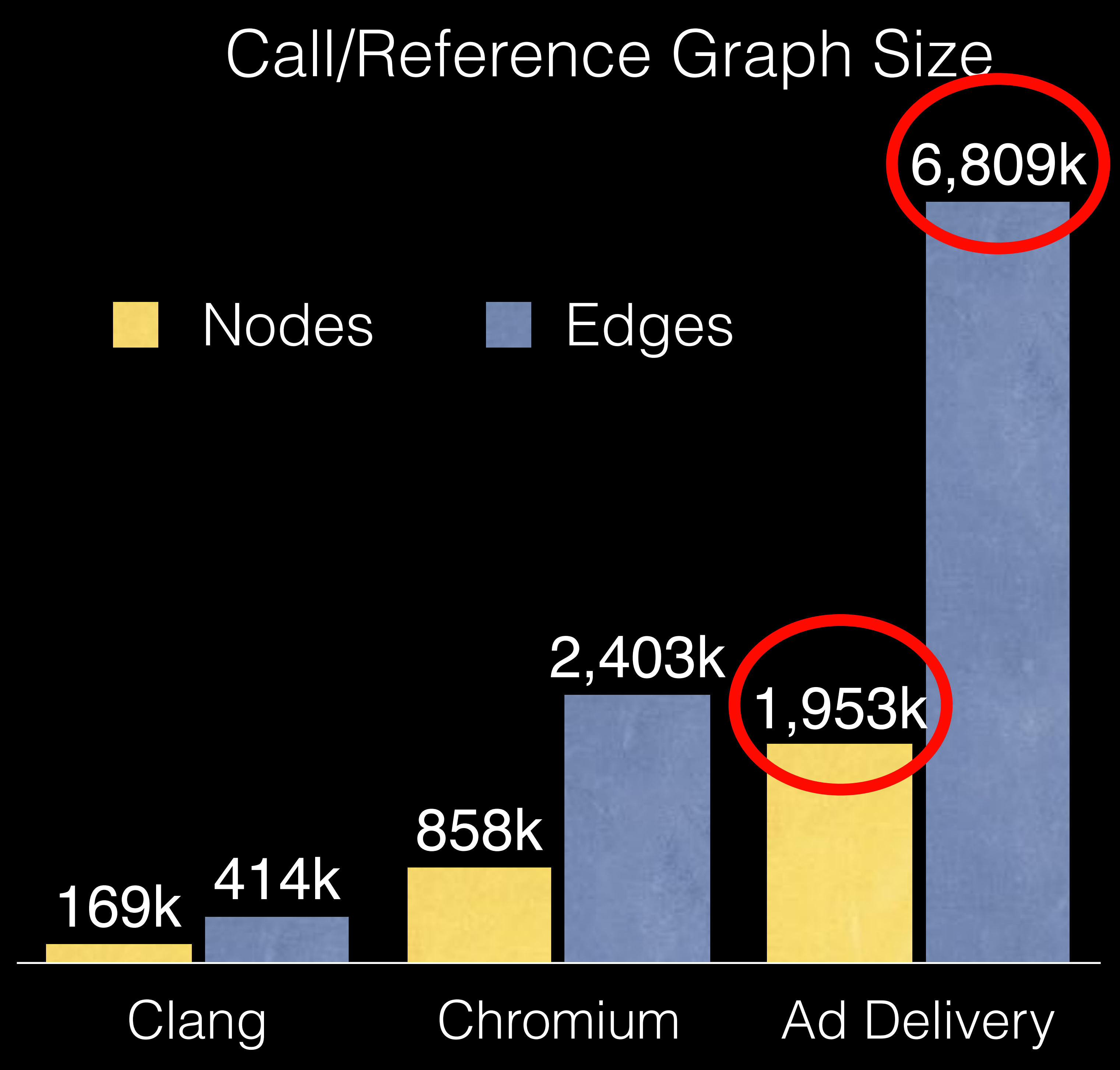
1. WPA: Serial part that makes IPA and inlining decisions, rewrites partitioned IR
   - ➡ *Comparable Phase 2: Thin Link (both serial)*
2. LTRANS: Parallel backends performing inlining within each partition, plus usual optimizations and code generation
   - ➡ *Comparable to Phase 3: ThinLTO Backends (both parallel)*

Scaling with the Input Size

- LLVM/Clang - C/C++ Compiler
- Chromium - open-source web browser
- Ad Delivery - internal Google datacenter application

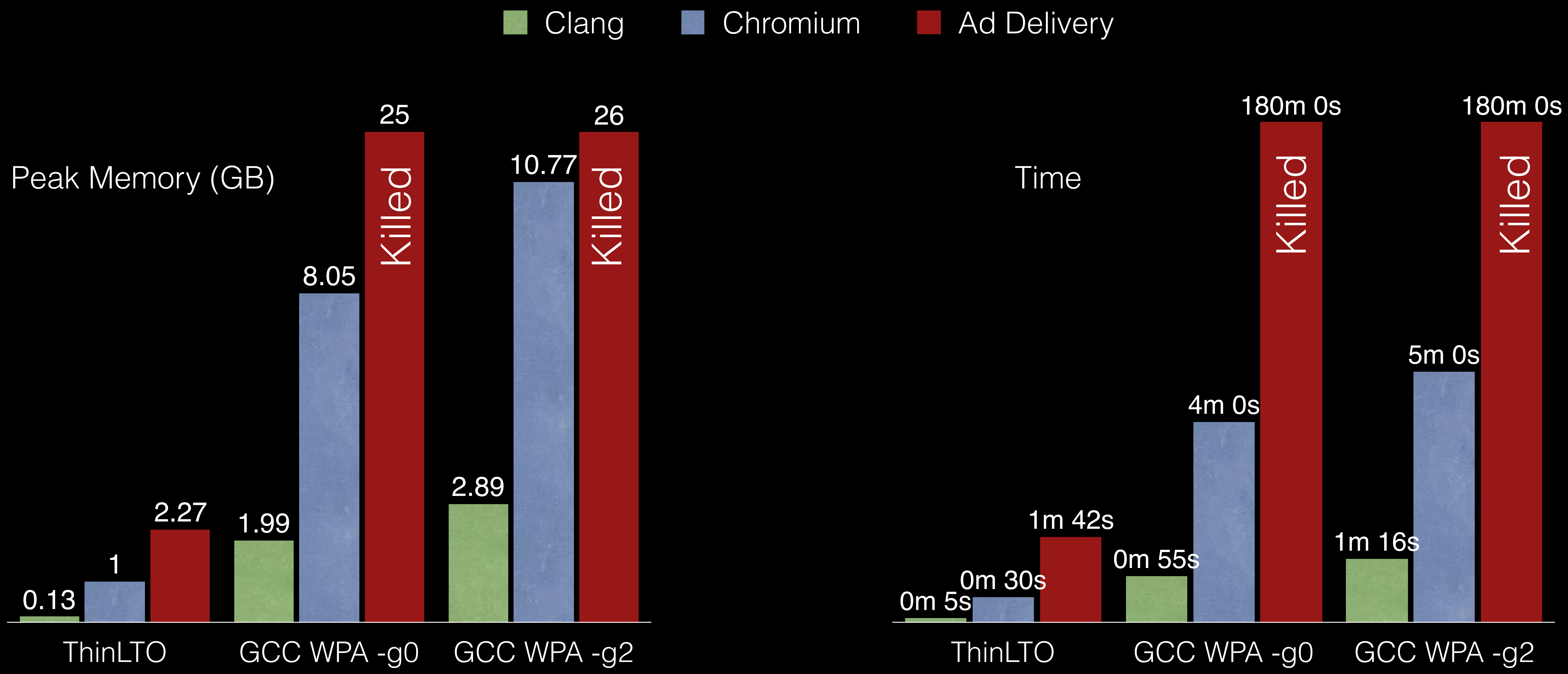






Serial Step Comparisons

Thin Link vs GCC WPA

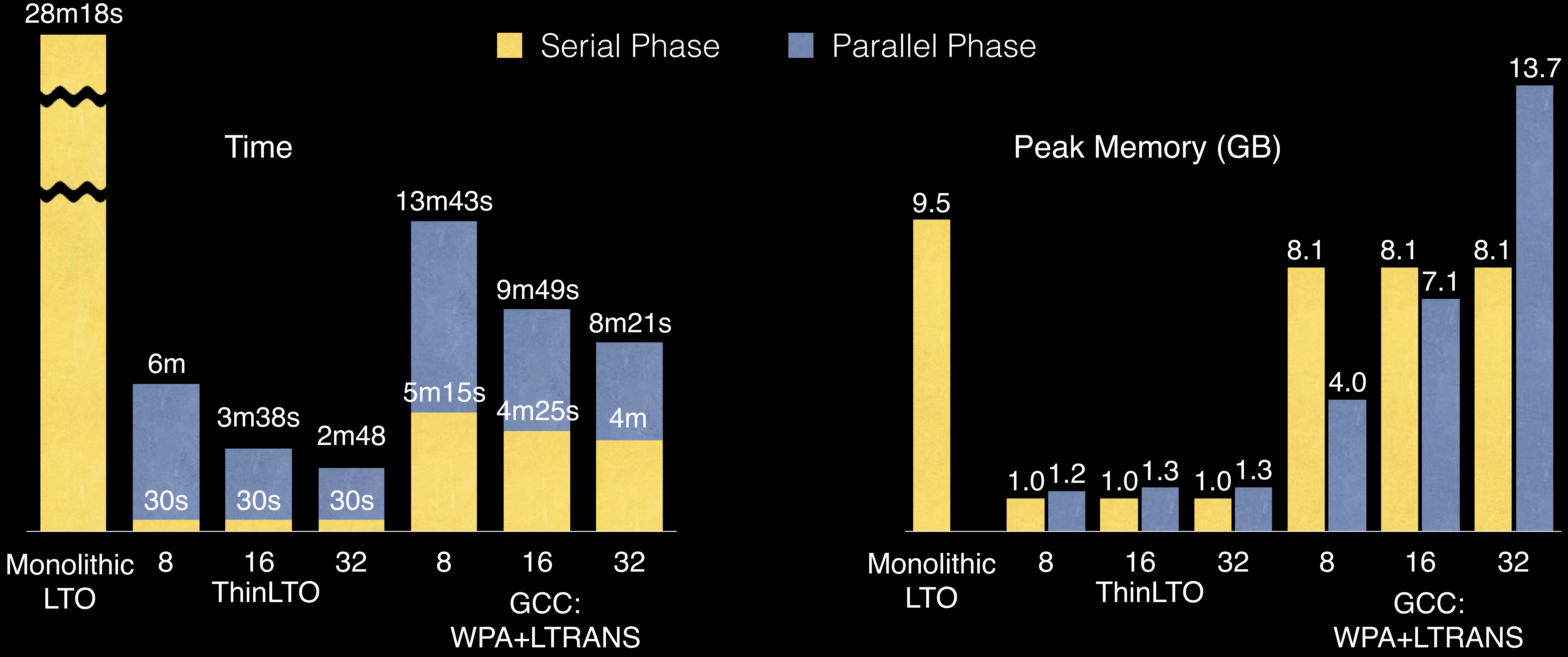


LLVM LTO doesn't complete Ad Delivery even without debug (-g0), killed after 2 hours and > 12GB.

Linux 16-core (32-logical) Intel Xeon E5-2690 @ 2.90GHz

Chromium Build Comparisons

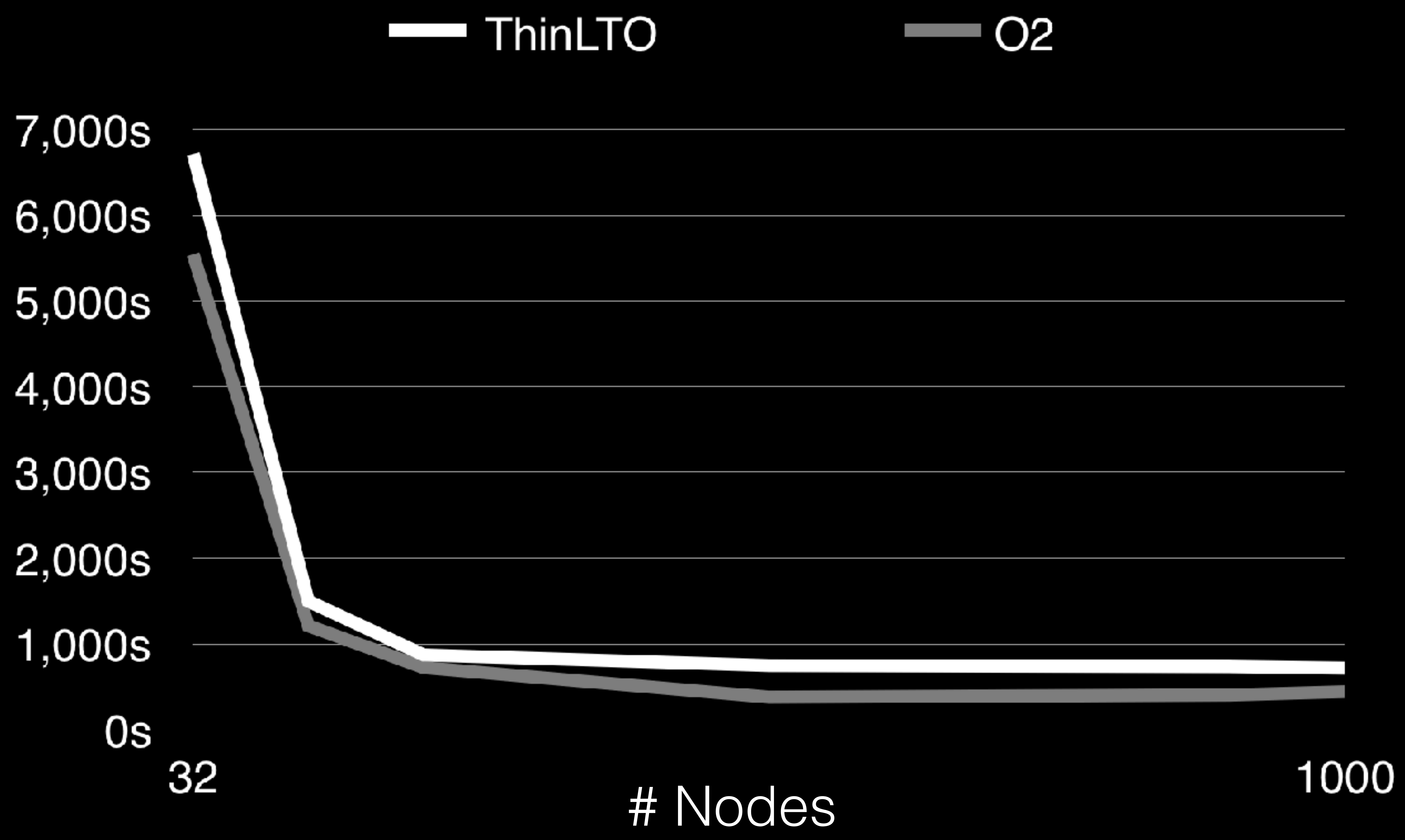
No Debug (-g0)



With Debug: Monolithic LTO crashes after >2h and >50GB mem

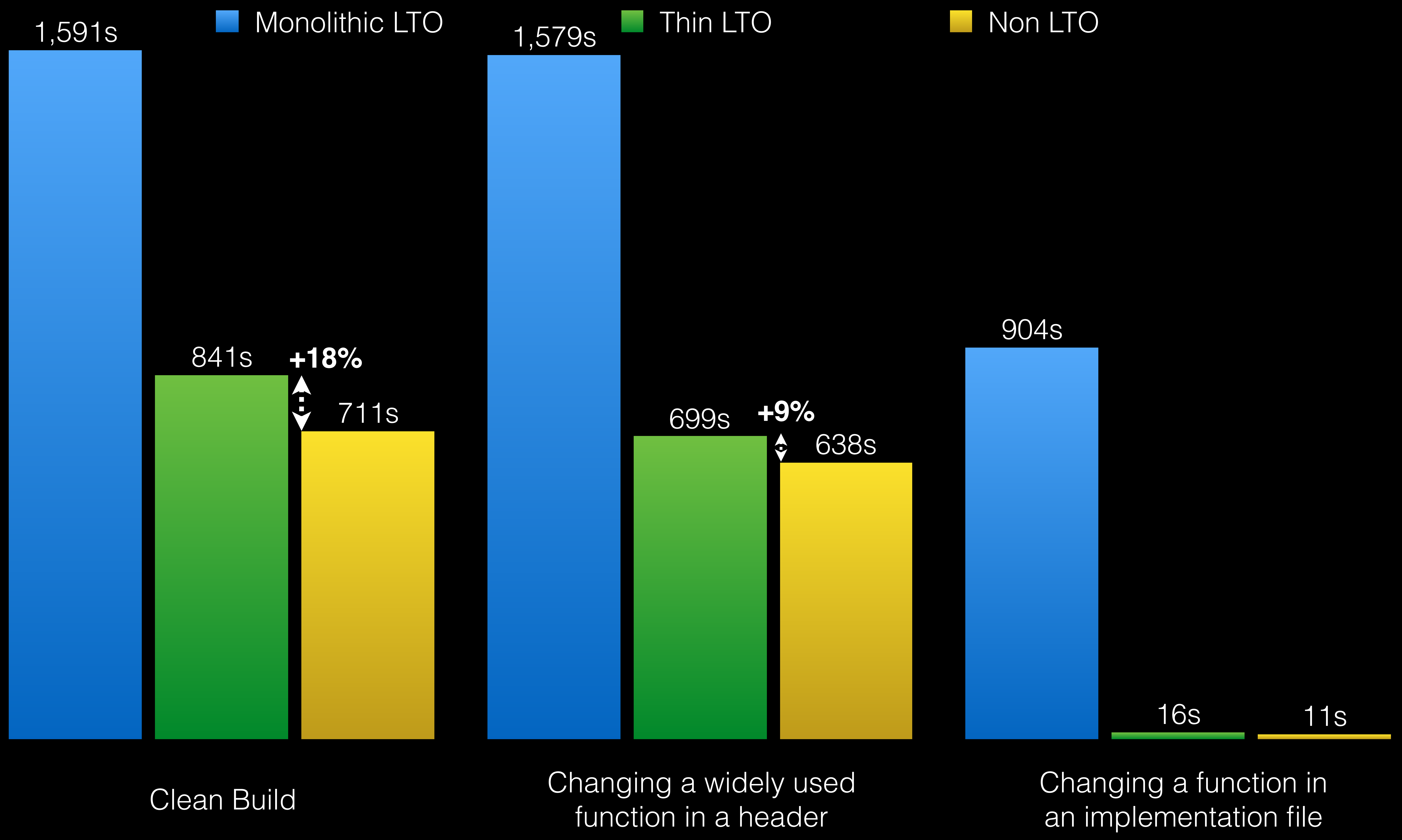
Linux 16-core (32-logical) Intel Xeon E5-2690 @ 2.90GHz

Distributed Build: Ad Delivery



Performance: **Incremental** Build Time for Clang-4.0

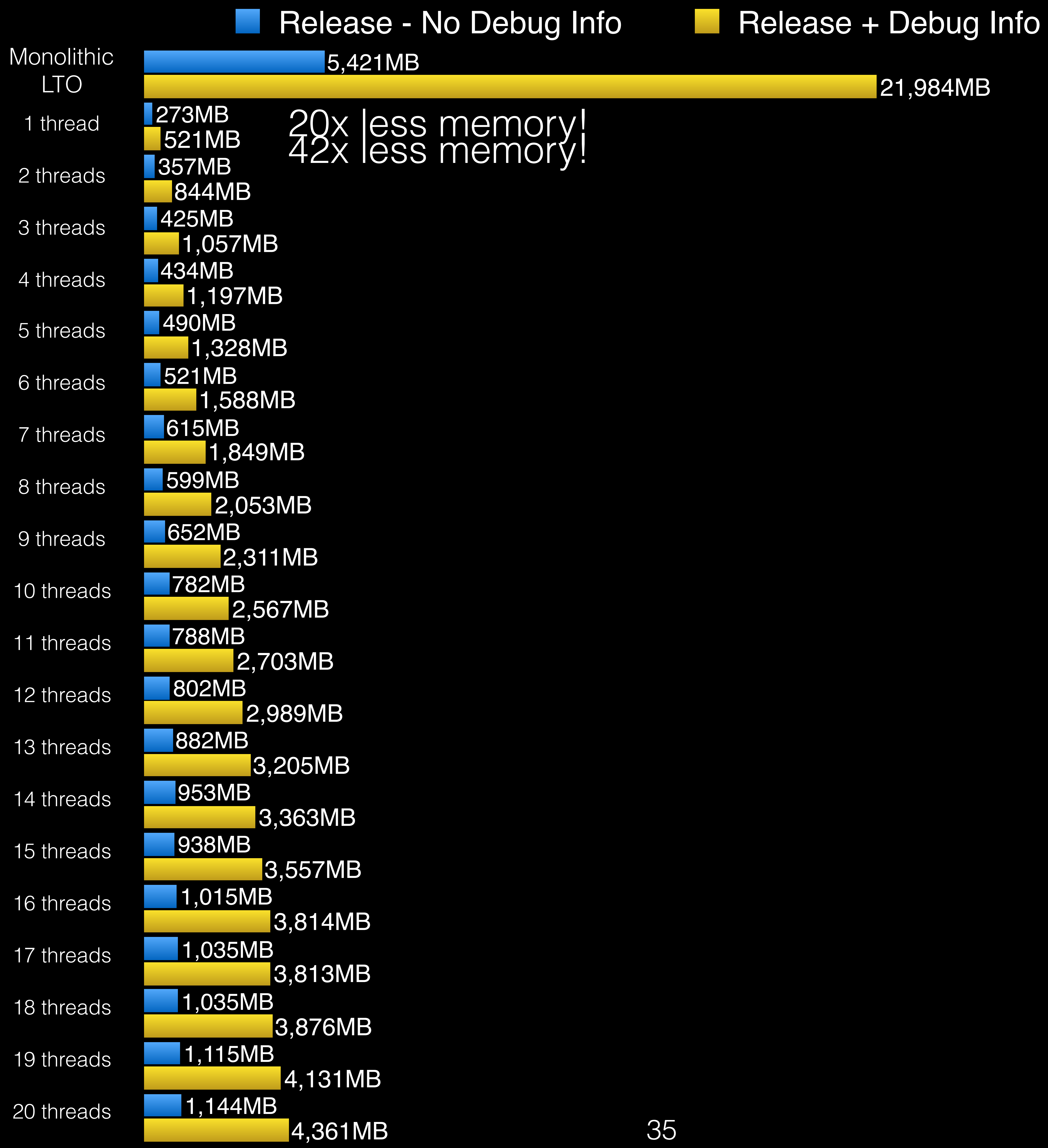
Time to run `ninja clang` ; all backends included



2013 MacPro - 12-cores E5-2697 @ 2.70GHz

Performance: Memory Consumption vs Parallelism

Link-Time Only
for clang-4.0

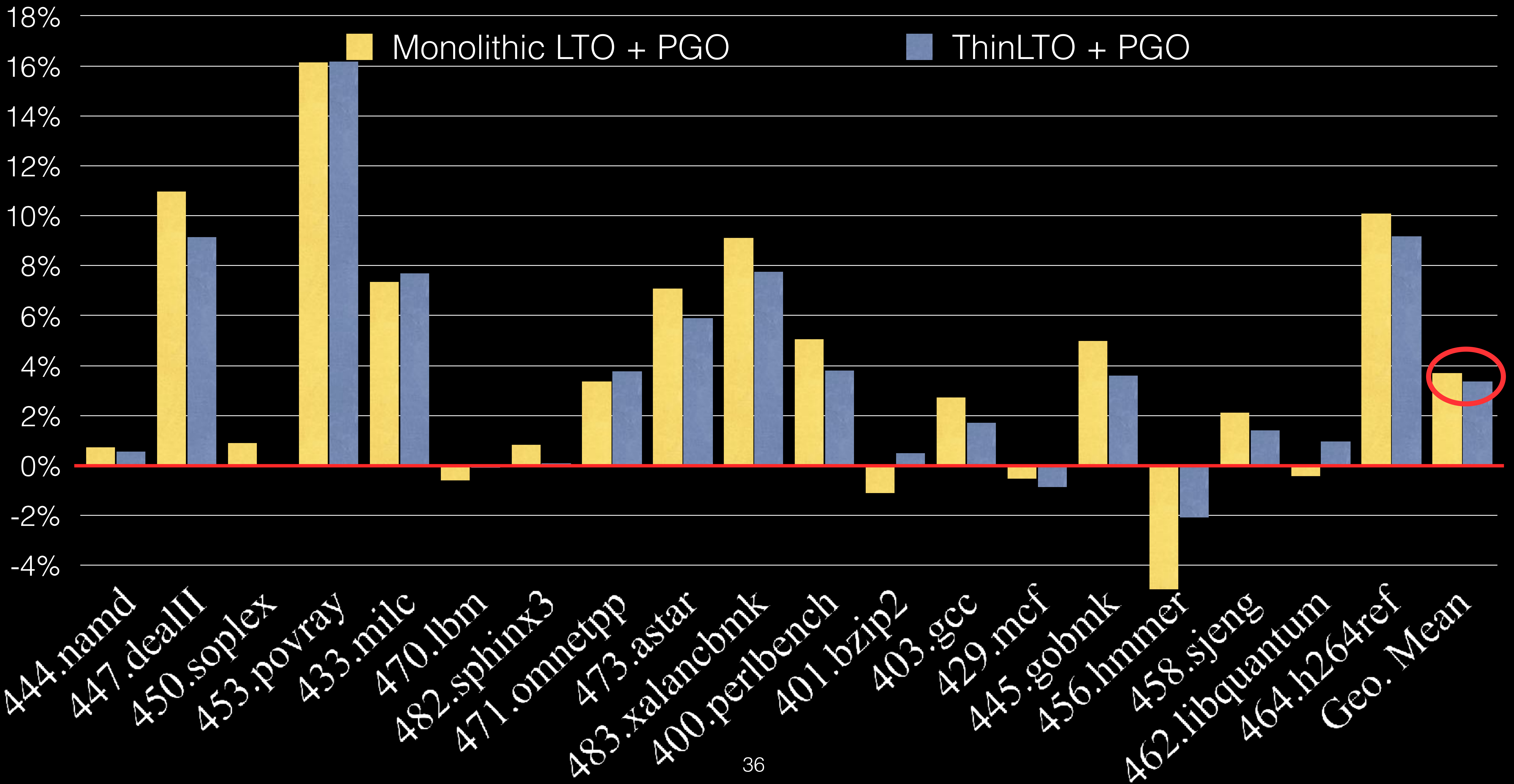


- 41MB per thread
- 192MB per thread

2013 MacPro - 12-cores E5-2697 @ 2.70GHz

Run-time Performance: SPEC cpu2006

Improvement over -O2 (**all with PGO**)



Status

- Supported by multiple linkers:
  - gold (via LLVM gold-plugin): as of clang 3.9
  - ld64: as of Xcode 8
  - lld: as of clang 4.0
- For usage information see:

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

Implications for C++ Developers

- Common advice: Put definitions of small functions in headers (using *inline* keyword to avoid ODR errors)
  - \+ Enables inlining into callers
  - \- Compile and possibly codegen in every #include-ing translation unit

foo.h

```
inline void f() {
  …
}
```

foo.cc

```
# “./foo.h”
inline void f() {
  …
}
f();
```

file1.cc

```
# “./foo.h”
inline void f() {
  …
}
f();
```

file2.ii

```
# “./foo.h”
inline void f() {
  …
}
```

…

fileN.ii

```
# “./foo.h”
inline void f() {
  …
}
```

Implications for C++ Developers

- With ThinLTO: Function definitions can stay in implementation file (caveats)
  - Compile one copy (to IR), and codegen one copy (to object)
  - Imported and optimized only where needed

foo.h

```
void f();
```

foo.cc

```
#include "foo.h"
void f() {
  // some stuff
}
f();
```

file1.cc

```
#include "foo.h"

f();
available_externally
void f() {
  // some stuff
}
```

file2.cc

```
#include "foo.h"
```

. . .

fileN.cc

```
#include "foo.h"
```

Caveats (functions with vague linkage that need to be defined in header):

- Template functions (unless explicitly specialized)
- Functions with the *inline* keyword - just remove it

Future Work

+ Propagate other function properties (e.g. “does not modify its arguments”)
+ *Augment* the global reference graph edges with mod-ref, cst range, …
+ Ability to *move* function instead of copying when a single call site exists
+ Integrate ThinLTO backend parallelism with *make -j*

Conclusion

Scales as a regular non-LTO build

Performance close to Monolithic LTO

Incremental build is critical for day-to-day development!

On the path of replacing O2/O3 by default in production build