ThinLTO: Scalable and Incremental LTO (CGO'17)

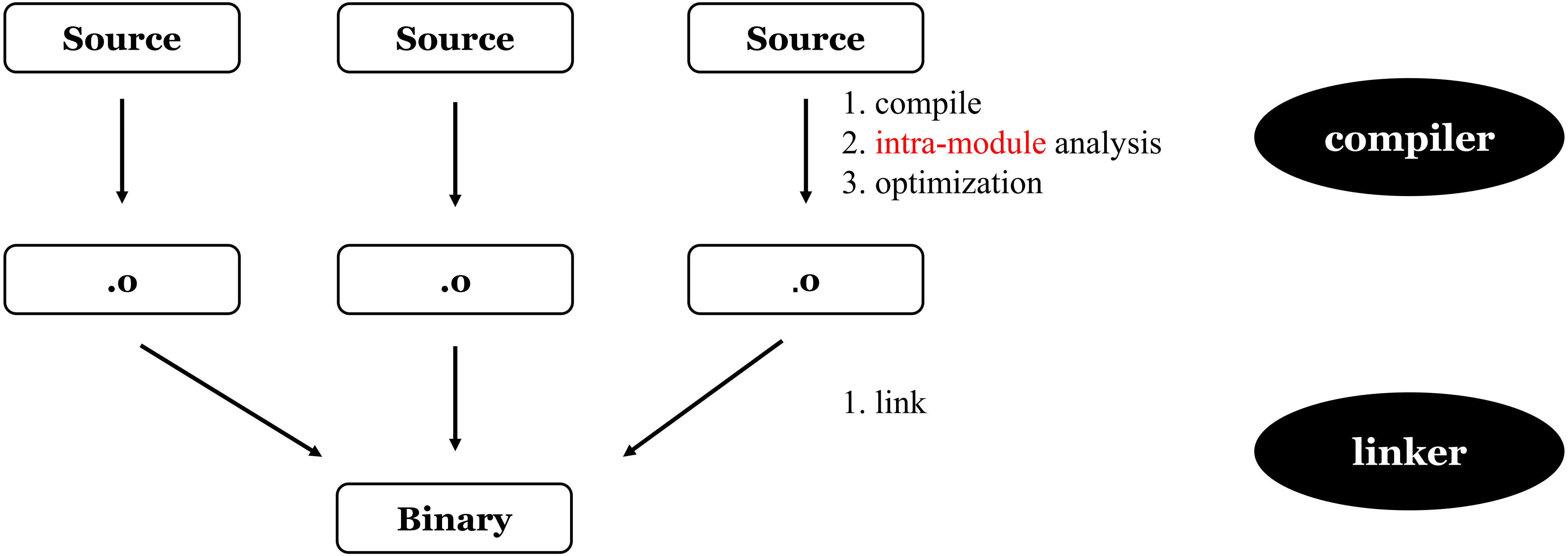
reading event

Nov. 4, 2021. Bowen Zhang

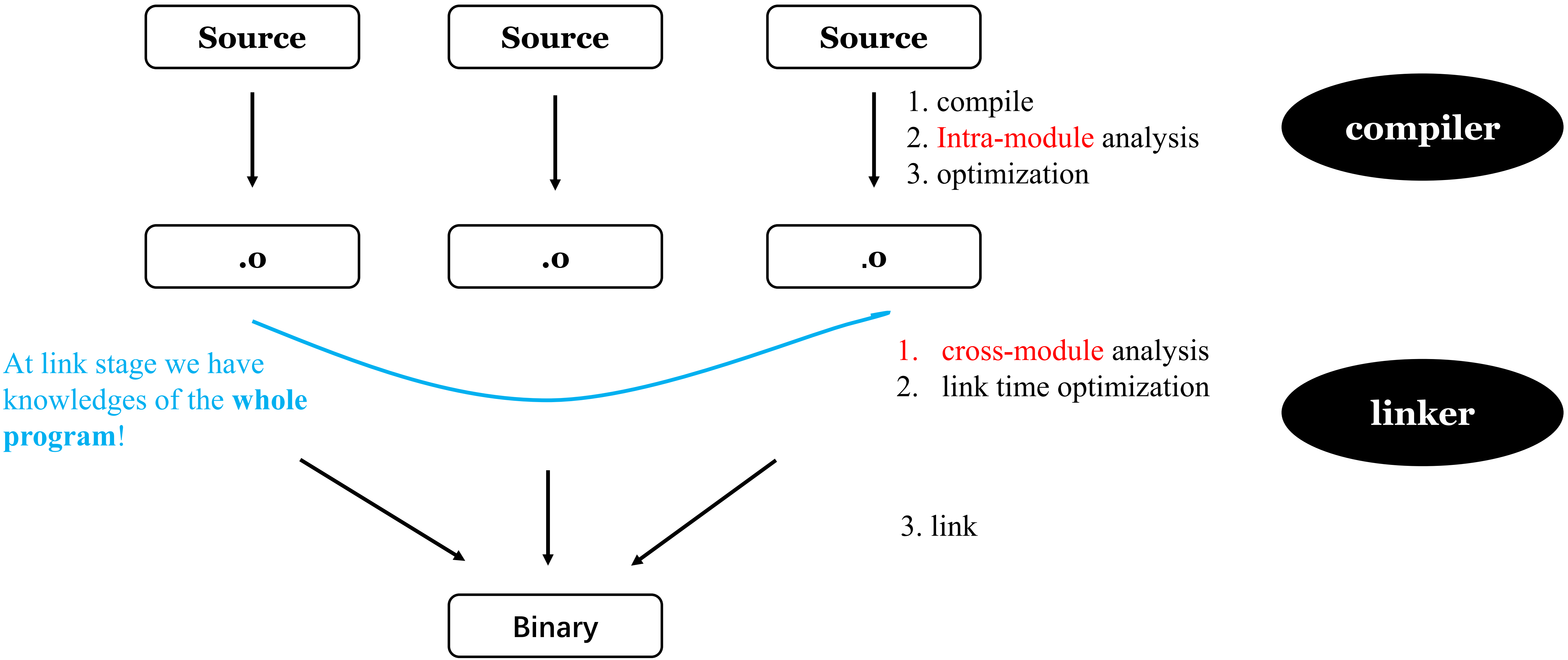
Contents

- Link Time Optimization(LTO)
  - What is LTO?
  - What benefits?
- Why LTO couldn’t be enabled by default?
- How ThinLTO overcomes this…
- Discussion

Traditional Build



Build with LTO



Benefits

Smaller Binary Size

- Main contribution
  - Dead Code Elimination
  - Constant Propagation

Better Performance

- 10% improvement is common.
- Main contribution: Inline
  - Usually performed with PGO(profile-guided optimization)

Binary Size

Can we reduce the code size?
-very hard

We don't know whether foo2 is called at some other places, so we can't eliminate this if-block.

A.c

```c
void foo4(void);

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;
}
```

Binary Size

Can we reduce the code size now?
-yes

main.c

```c
#include <stdio.h>

int foo1(void);

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


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

foo4 never used

(don’t need to link printf() into the binary too!)

A.c

```c
void foo4(void);

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;
}
```

foo2 never used

foo3 never used

i < 0 always false

data = 42

Performance

Do we reach the end?
-no

main.c

```c
#include <stdio.h>

int foo1(void);

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


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

Inline

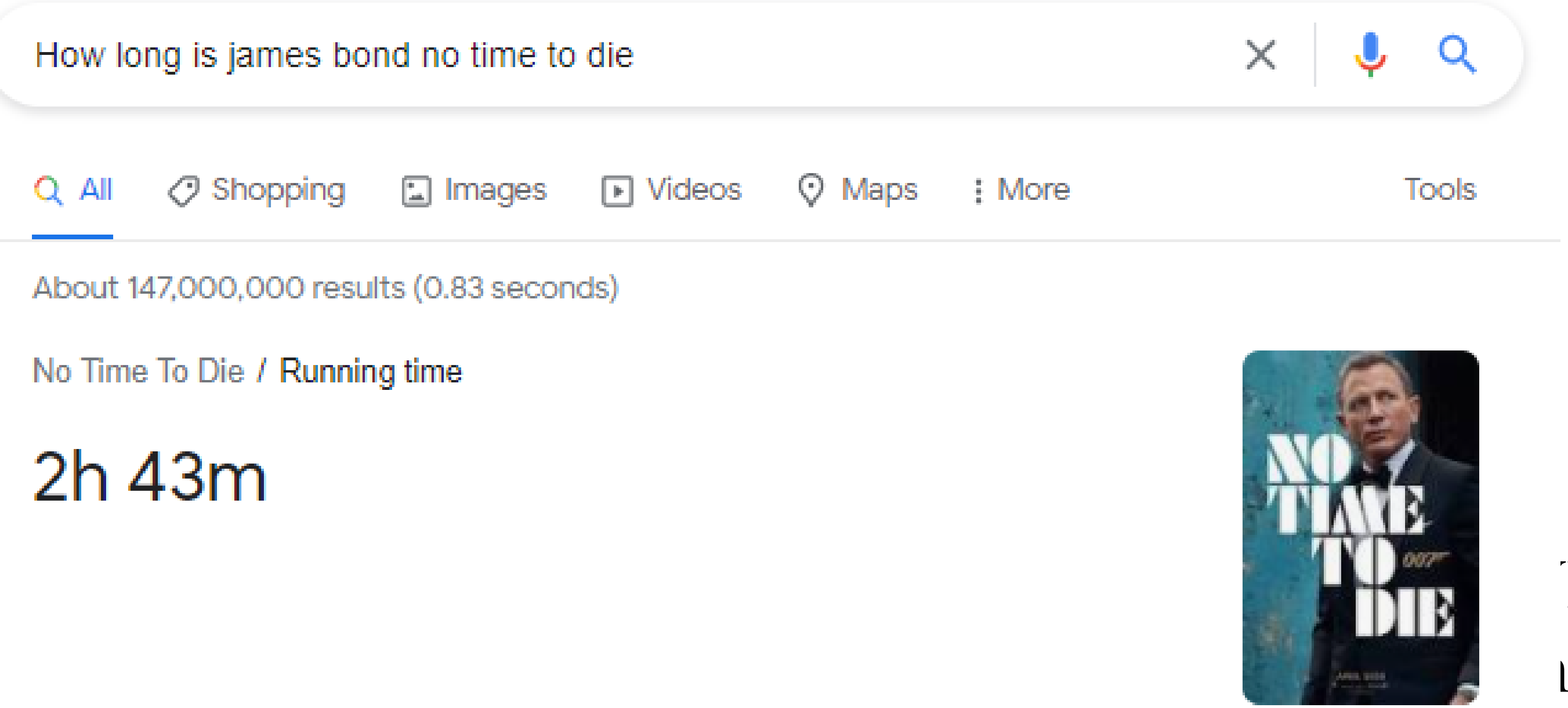
A.c

```c
void foo4(void);

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;
}
```

try compile this example with LTO yourself
clang –c main.c A.c -flto
clang –o main main.o A.o -flto

Can't enable LTO by default

GCC's LTO(–g0)



M's LTO can't
finish this task.

which has 2X call graph nodes, and 3X call graph edges, compared with Chromium

ThinLTO: Why?

scalable & low memory lean

GCC’s LTO(–g0)
ThinLTO

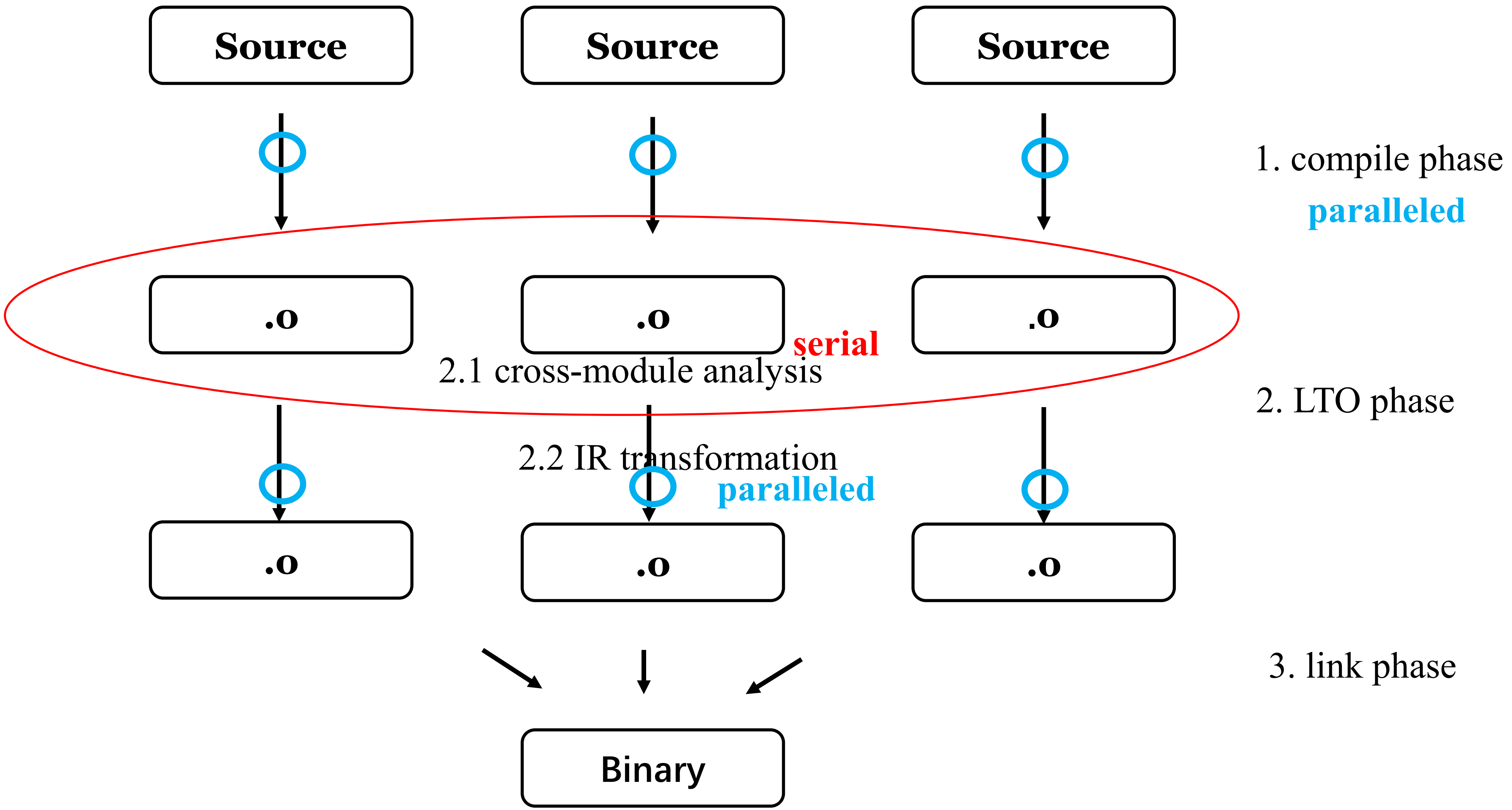
clang	55 s 5 s	2 GB 0.13 GB
chromium	4 mins 30 s	8 GB 1 GB
Ad Delivery	3 hours 102s	>25 GB 2.27 GB

**Google's ad recommendation application which has 2X call graph nodes, and 3X call graph edges, compared with Chromium*

Previous Design of LTO

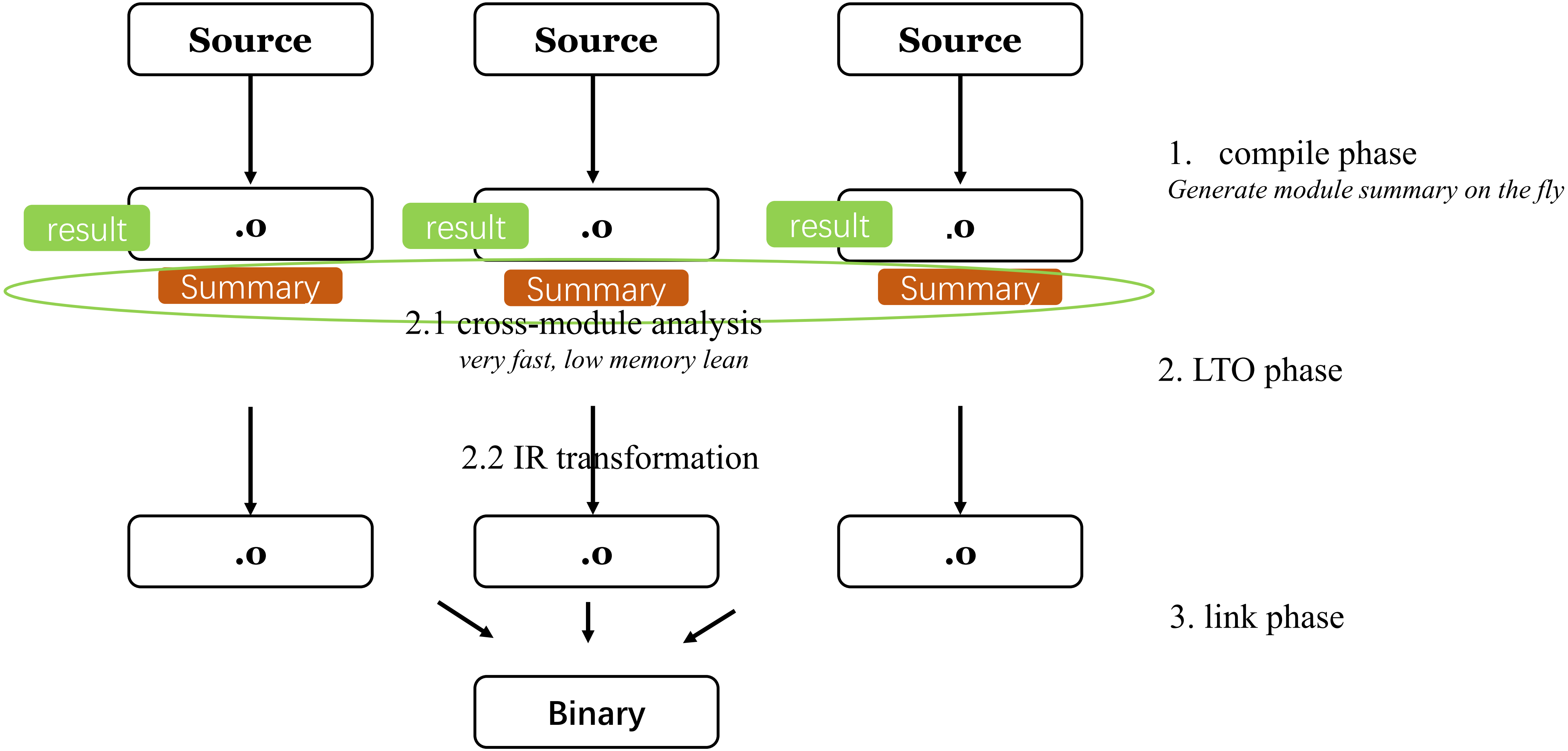
GCC, Clang(<3.9), and others

What's the bottleneck?



ThinLTO Design

Summary-based LTO



Represent Module Summary?

A naïve inline opt under ThinLTO

main.c

```
1   int foo(int);
2   int bar(int);
3
4   int main() {
5       int x = 42;
6
7       if(...) {
8           x = bar(x);
9       } else {
10          x = foo(x);
11      }
12      return x;
13  }
```

hotness	
foo() at line 10	5
bar() at line 8	30

score	
foo() at line 10	20
bar() at line 8	10

A.c

```
1   int foo(int x) {
2       return 2 + x;
3   }
4
5   int bar(int y) {
6       return (x * 2) + (x * 8);
7   }
```

weigh	
foo	100
bar	300

Our Inline heuristic:
A function call **f** at call site **c** has inline score:
score(f, c) = weigh(f) / hotness(c)
If score < 15 then we inline f at this call site.

**weigh(f) is the 100* (total operations in f).*
**hotness(c) is the called frequency recorded by profiling*

ThinLTO is a framework, not specific optimization algorithms

Design our own optimizations in ThinLTO framework

- Compile phase: collect and add information to module-summary.
    - */lib/Analysis/ModuleSummaryAnalysis.cpp*
- Cross-module analysis: gather module-summary and generate analysis result for each module.
    - *lib/LTO/ThinLTOCodeGenerator.cpp::ProcessThinLTOModule()*
- IR Transformation: For each module, perform optimizations and transformations **individually** and parallelly.
    - */lib/LTO/LTOBackend.cpp*

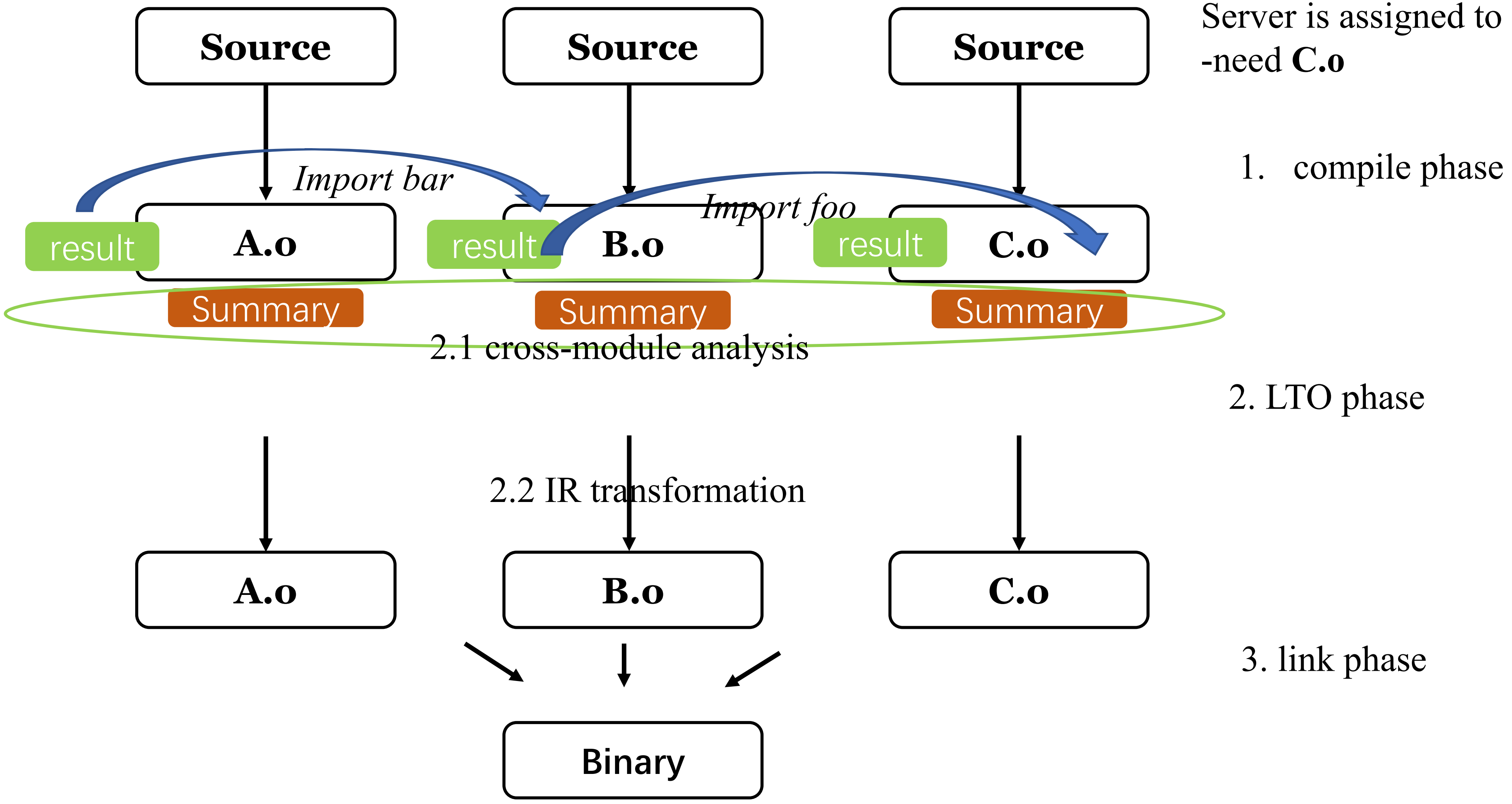
Distributed Build Supports

contain “import info” in analysis result

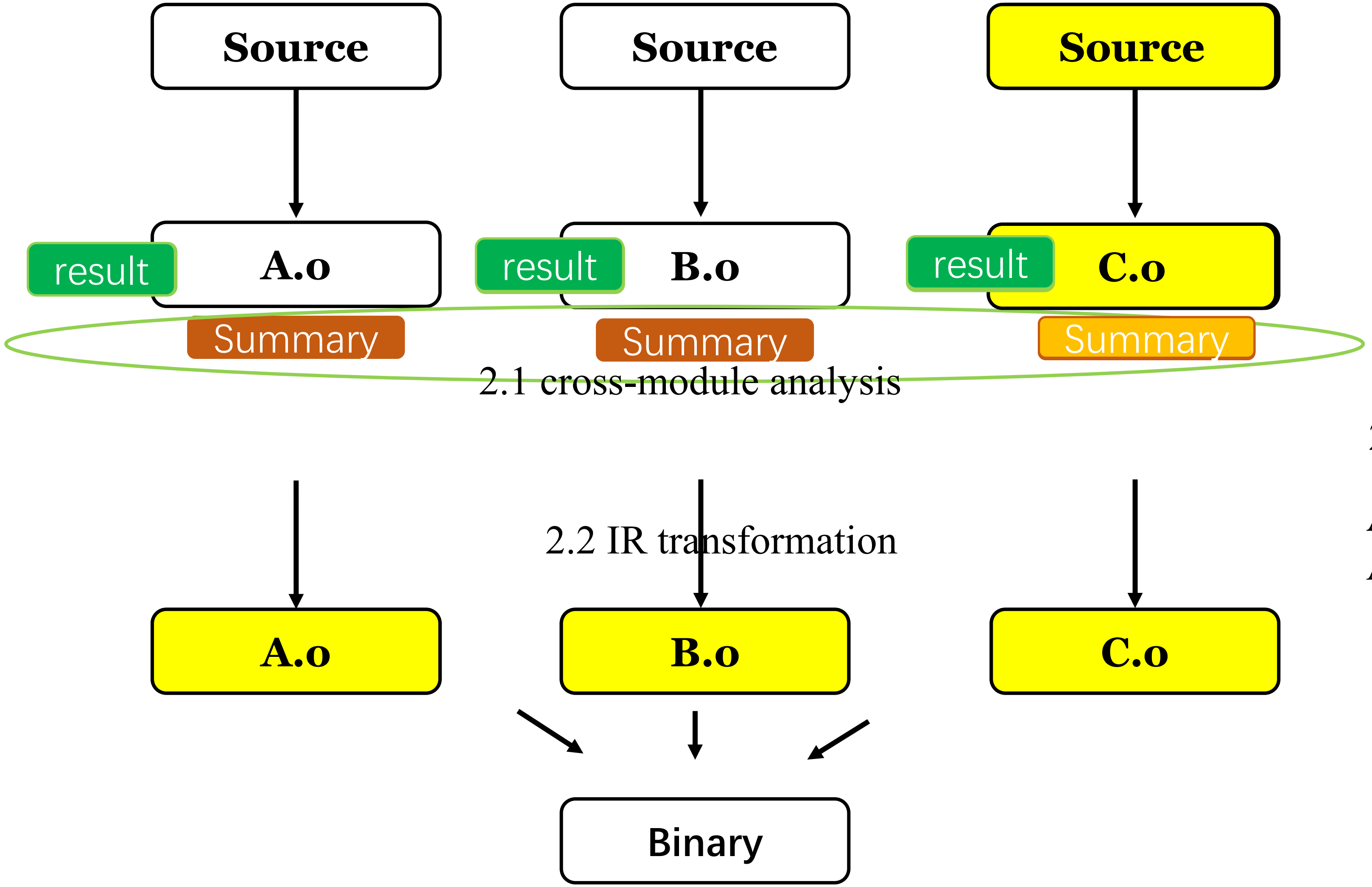
Server is assigned to do phase2.2 for **A.o**
-need **A.o** + **B.o**

Server is assigned to do phase2.2 for **B.o**
-need **B.o** + **C.o**

Server is assigned to do phase2.2 for **C.o**
-need **C.o**



Incremental Build Support



1. compile phase

Natually incremental(make)

maintain a cache system:

Source → .o file & summary

2. LTO phase

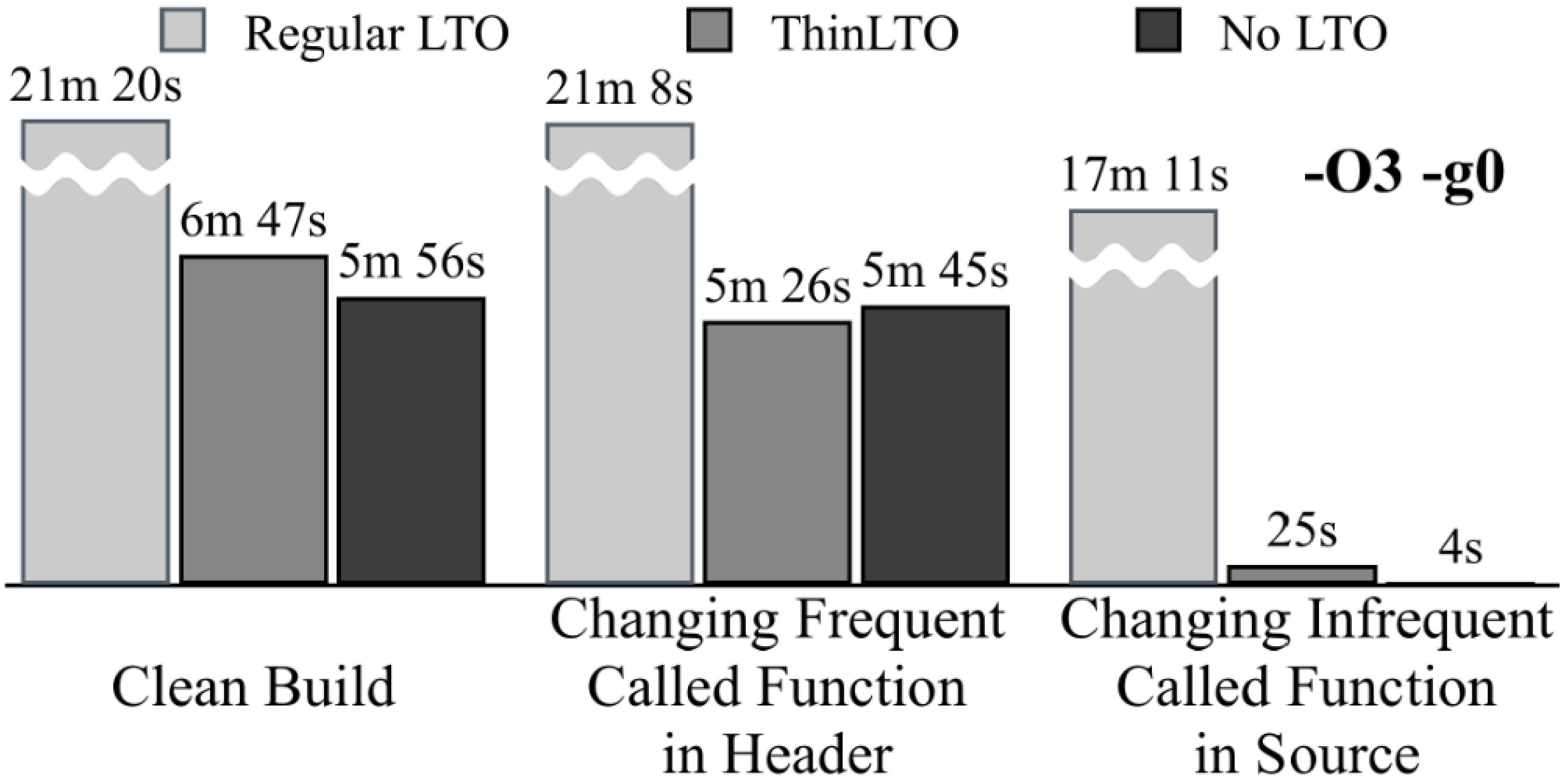
A cache system

Analysis-result → .o file

3. link phase

Incremental Build Support:

Not always so helpful?



Discussion

- Incremental build
  - Can we reduce redundant computation even when frequent called function is modified?

Discussion

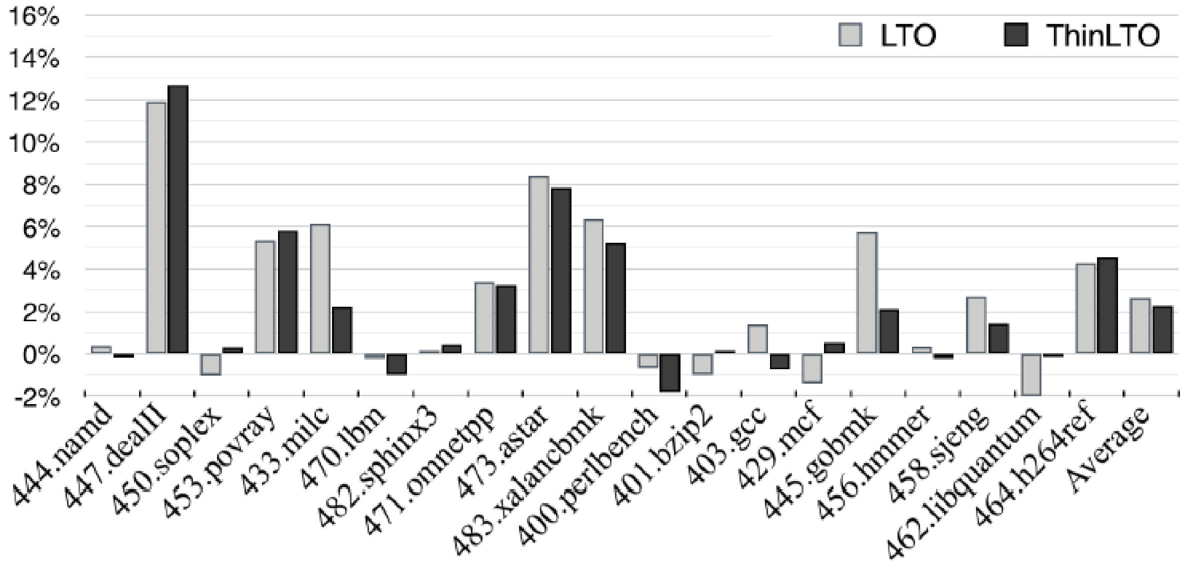
Whole program analysis

Optimize and transform for each .o file

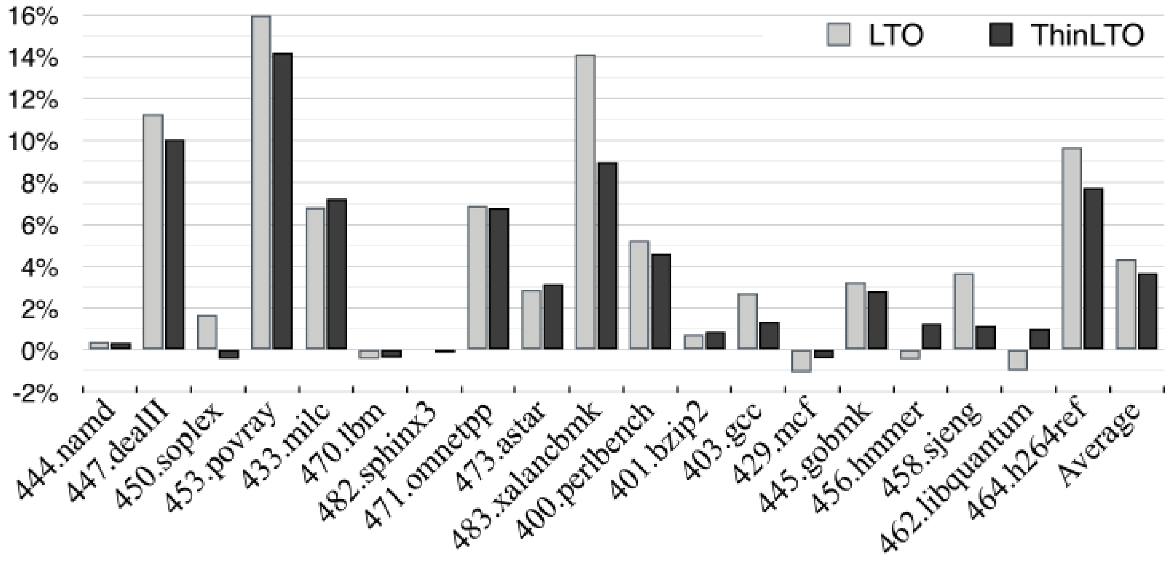
Summary based LTO is like one-step optimization

Limitation: Optimize result (make program run faster)

- ThinLTO's optimize result on SPEC is just "as good" as LTO.
  - ThinLTO doesn't do **aggressive** whole program analysis.



(a) Without Profile-Guided Optimization (PGO).



(b) With Profile-Guided Optimization (PGO).

Discussion

- Optimization limitation: cont

main.c

```c
#include <stdio.h>

int foo1(void);

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


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

A.c

```c
void foo4(void);

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;
}
```

Consider that **i** and **foo3** are not static, and are located in other files.

Can ThinLTO handle this?

Discussion

- Consider static analysis after LTO.
  - After LTO, code is smaller while semantic is the same
  - ThinLTO is fast and we can afford this cost.

Thank you!

Links

- ThinLTO: https://dl.acm.org/doi/10.5555/3049832.3049845
- LLVM doc: https://clang.llvm.org/docs/ThinLTO.html
- Talk at cpp-conf: https://youtu.be/p9nH2vZ2mNo
- GCC's LTO: https://arxiv.org/abs/1010.2196