ThinLTO

A Framework for Scalable and Incremental Link-Time Optimization

Teresa Johnson  Mehdi Amini  Xinliang David Li
ThinLTO
Towards Always-Enabled LTO
A Framework for Scalable and Incremental Link-Time Optimization

Teresa Johnson    Mehdi Amini    Xinliang David Li
LLVM LTO: in a Nutshell

```cpp
int test1(int a);
int main() {
    return test1(42);
}
```

```cpp
int test1(int a) {
    // Do stuff
    ... return test2(a+1);
    int test1_foo(int bar) {
        return bar-1;
    }
}
```

```cpp
int test2(int);
int test2(int b) {
    // Do stuff
    return test1_foo(b + 1);
}
```
LLVM LTO: in a Nutshell

```
1 int test1(int a);
2 int main() {
3   return test1(42);
4 }
```

```
1 int test2(int);
2 int test1(int a) {
3   // Do stuff
4     // ...
5   return test2(a+1);
6 }
7 int test1_foo(int bar) {
8   return bar-1;
9 }
```

```
1 int test1_foo(int);
2 int test2(int b) {
3   // Do stuff
4   return test1_foo(b + 1);
5 }
```
LLVM LTO: in a Nutshell

```cpp
int test1(int a);
int main() {
    return test1(42);
}
```

```cpp
int test2(int b) {
    // Do stuff
    // ...
    return test2(a+1);
}
int test1_foo(int bar) {
    return bar-1;
}
```

```
clang -cc1 -flto
```

```
main.o
test1.o
test2.o
```

.o files are generated, but they are actually raw bitcode files

static archive will contain these bitcode files
LLVM LTO: in a Nutshell
Highly parallel frontend processing + initial optimizations
Highly parallel frontend processing + initial optimizations
LLVM LTO: in a Nutshell

Highly parallel frontend processing + initial optimizations

Frontend Linker

LLVM Bitcode
LLVM LTO: in a Nutshell

Highly parallel frontend processing + initial optimizations

LLVM as a linker-plugin (libLTO.dylib or LLVMgold.so)
LLVM LTO: in a Nutshell

- Highly parallel frontend processing + initial optimizations
- Link all bitcode in one single Module

Monolithic LTO Implementation

LLVM as a linker-plugin (libLTO.dylib or LLVMgold.so)

.bitcode
LLVM LTO: in a Nutshell

Highly parallel frontend processing + initial optimizations

Link all bitcode in one single Module

Monolithic LTO Implementation

Single-threaded very boring usual usual optimizations
LLVM LTO: in a Nutshell

Highly parallel frontend processing + initial optimizations

Link all bitcode in one single Module

Monolithic LTO Implementation

Single-threaded very boring usual optimizations

Potentially threaded CodeGen

LLVM as a linker-plugin (libLTO.dylib or LLVMgold.so)

.o .o .o .o .o .o .o .o

Frontend Linker

LLVM Bitcode

Optimizer / Inlining

CodeGen

Traditional Linking
Why LTO?

Benefits
Why LTO?

Benefits

• Binary size: inherent dead-stripping and auto-hidden visibility \textit{via} internalization.
Why LTO?

Benefits

• Binary size: inherent dead-stripping and auto-hidden visibility via internalization.
• Performance! 10% boost is common.
  ➔ Removes module optimization boundaries via Cross-Module Optimization (CMO)
  ➔ Most of benefit comes from cross-module inlining
Why LTO?

Benefits

- Binary size: inherent dead-stripping and auto-hidden visibility via internalization.
- Performance! 10% boost is common.
  - Removes module optimization boundaries via Cross-Module Optimization (CMO)
  - Most of benefit comes from cross-module inlining

<table>
<thead>
<tr>
<th>Performance Improvements - Execution Time</th>
<th>Δ</th>
<th>Previous</th>
<th>Current</th>
<th>σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>SingleSource/Benchmarks/Shootout_C++/objlnst</td>
<td>-96.80%</td>
<td>0.0937</td>
<td>0.0030</td>
<td>-</td>
</tr>
<tr>
<td>SingleSource/Benchmarks/Dhrystone/lldry</td>
<td>-93.17%</td>
<td>0.0893</td>
<td>0.0061</td>
<td>0.0002</td>
</tr>
<tr>
<td>SingleSource/Benchmarks/Dhrystone/dry</td>
<td>-79.21%</td>
<td>0.0534</td>
<td>0.0111</td>
<td>0.0000</td>
</tr>
<tr>
<td>SingleSource/Benchmarks/Misc/matmul_f64_4x4</td>
<td>-52.78%</td>
<td>0.0199</td>
<td>0.0094</td>
<td>0.0008</td>
</tr>
<tr>
<td>SingleSource/UnitTests/Vector/build2</td>
<td>-48.56%</td>
<td>0.0421</td>
<td>0.0225</td>
<td>0.0002</td>
</tr>
<tr>
<td>MultiSource/Benchmarks/tramp3d-v4/tramp3d-v4</td>
<td>-35.45%</td>
<td>0.1839</td>
<td>0.1187</td>
<td>0.0026</td>
</tr>
<tr>
<td>SingleSource/Benchmarks/Misc/ltopa-8</td>
<td>-33.50%</td>
<td>0.0403</td>
<td>0.0258</td>
<td>0.0001</td>
</tr>
<tr>
<td>MultiSource/Benchmarks/Okken/bh/bh</td>
<td>-31.00%</td>
<td>0.1412</td>
<td>0.0973</td>
<td>0.0002</td>
</tr>
<tr>
<td>MultiSource/Benchmarks/Okken/power/power</td>
<td>-30.45%</td>
<td>0.0867</td>
<td>0.0603</td>
<td>0.0000</td>
</tr>
</tbody>
</table>
Why LTO?

Benefits

• Binary size: inherent dead-stripping and auto-hidden visibility via internalization.
• Performance! 10% boost is common.

➤ Removes module optimization boundaries via Cross-Module Optimization (CMO)
➤ Most of benefit comes from cross-module inlining
Why LTO?

Benefits

- Binary size: inherent dead-stripping and auto-hidden visibility via internalization.
- Performance! 10% boost is common.
  - Removes module optimization boundaries via Cross-Module Optimization (CMO)
  - Most of benefit comes from cross-module inlining

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.**
```c
#include <stdio.h>

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

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

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

```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;
}
```
```c
#include <stdio.h>

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

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

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

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

// 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;
    }
}
```
Example

```c
#include <stdio.h>

int foo1(int x)
{
    printf("Hi\n");
    return x + 42;
}

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

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

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();
}
```
Example
Example
Example
Example

```c
#include <stdio.h>

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

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

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

```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 main(void) {
    int data = 0;
    if (i < 0) {
        data = foo3();
    }
    data = data + 42;
    return data;
}
```

```c
static int foo3() {
    foo4();
    return 10;
}

static int foo1(void) {
    int data = 0;
    return foo1();
}

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

```
#include <stdio.h>

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

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

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

```
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;
}
```
Example

```c
#include <stdio.h>

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

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

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

```c
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;
}
```
Example
Example

Optimization across module boundaries!

+ Linker Information
Monolithic LTO: No Free Lunch!
Monolithic LTO: No Free Lunch!

LTO adoption is still low after >10 years of existence: why?
Monolithic LTO: No Free Lunch!

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

- Slow: inherently serial / can’t be distributed

```
' time ninja clang'
```

252s 970s
Monolithic LTO  Non LTO
Monolithic LTO: No Free Lunch!

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

- Slow: inherently serial / can’t be distributed
  
  `time ninja clang`

  `4s`  
  252s  
  970s

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

  `1000`  
  252s  
  970s  
  734s  
  4s
Monolithic LTO: No Free Lunch!

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

- Slow: inherently serial / can’t be distributed

```
$time ninja clang
```

```
4s
```

- Monolithic LTO: 252s
- Non LTO: 4s
- 970s

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

```
`ninja clang`
```

```
734s
```

https://www.xkcd.com/303/
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 a single bitcode file in memory.

```
$ time ninja clang
```

<table>
<thead>
<tr>
<th></th>
<th>Monolithic LTO</th>
<th>Non LTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>970s</td>
<td>734s</td>
</tr>
<tr>
<td>Time (incremental)</td>
<td>252s</td>
<td>4s</td>
</tr>
</tbody>
</table>

https://www.xkcd.com/303/
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 a single bitcode file in memory.

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

- LLVM 3.6
  - `time ninja clang`
  - 4s
  - 734s

Monolithic LTO
Non LTO

https://www.xkcd.com/303/
Monolithic LTO: No Free Lunch!

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

- Slow: inherently serial / can’t be distributed

```
$time ninja clang
```

\[970s\]

- 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 a single bitcode file in memory.

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

- LLVM 3.6
- LLVM 3.7
- LLVM 3.8
- LLVM 3.9

https://www.xkcd.com/303/
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 a single bitcode file in memory.

```
$time ninja clang
```

<table>
<thead>
<tr>
<th>LLVM Version</th>
<th>Monolithic LTO</th>
<th>Non LTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLVM 3.6</td>
<td>252s</td>
<td>4s</td>
</tr>
<tr>
<td>LLVM 3.7</td>
<td>252s</td>
<td>4s</td>
</tr>
<tr>
<td>LLVM 3.8</td>
<td>970s</td>
<td>734s</td>
</tr>
<tr>
<td>LLVM 3.9</td>
<td>970s</td>
<td>734s</td>
</tr>
</tbody>
</table>

```

Dead end: linking Chromium with debug info still crashes now after >2h and >50GB mem

```
https://www.xkcd.com/303/
```
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 a single bitcode file in memory.
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 a single bitcode file in memory.

➤ Parallel
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 a single bitcode file in memory.

 ➤ Parallel

 ➤ Incremental
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 a single bitcode file in memory.  ➡️ Memory lean
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 a single bitcode file in memory.
  ➤ Memory lean

To fulfill this goal of LTO always enabled, we need a solution designed around these three key features!
ThinLTO: Design
ThinLTO: Design

• Designed from the start for huge (Google scale) applications
ThinLTO: Design

- Designed from the start for huge (Google scale) applications
- Fully parallel compile step and backends → enables distributed builds

Fully-parallel frontend processing + initial optimizations

Fully-parallel (very boring) usual optimizations and CodeGen
ThinLTO: Design

- Designed from the start for huge (Google scale) applications
- Fully parallel compile step and backends ➞ enables distributed builds

**Compile**

```
(bc) (bc) (bc) (bc) (bc) (bc) (bc) (bc)
```

**Thin Link**

- Fully-parallel frontend processing + initial optimizations

**Parallel Backends**

```
```

**Thin** serial synchronization step

- Fully-parallel (very boring) usual optimizations and CodeGen

**Traditional Linking**
ThinLTO: Design

- Designed from the start for huge (Google scale) applications
- Fully parallel compile step and backends ➞ enables distributed builds
- Module is unit of compilation ➞ enables incremental builds

Fully-parallel frontend processing + initial optimizations

Thin serial synchronization step

Fully-parallel (very boring) usual optimizations and CodeGen
ThinLTO: Design

- Designed from the start for huge (Google scale) applications
- Fully parallel compile step and backends \(\rightarrow\) enables distributed builds
- Module is unit of compilation \(\rightarrow\) enables incremental builds
- Only perform profitable cross module optimization into each module \(\rightarrow\) memory scaling

Fully-parallel frontend processing + initial optimizations

Thin serial synchronization step

Fully-parallel (very boring) usual optimizations and CodeGen
ThinLTO: Overview
ThinLTO: Overview

- Fully-parallel frontend processing
- + initial optimizations

[Diagram showing process with .bc files]
ThinLTO: Overview

Phase 1: Compile

Extra per-function summary information are generated “on the side”

Fully-parallel frontend processing + initial optimizations
ThinLTO: Overview

Phase 1: Compile

- Fully-parallel frontend processing
- + initial optimizations
- Extra per-function summary information are generated “on the side”
ThinLTO: Overview

Phase 1: Compile

- Compile

Phase 2: Thin Link

- Thin Link

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.

No need to parse the IR
ThinLTO: Overview

Phase 1: Compile

- Fully-parallel frontend processing + initial optimizations
- Extra per-function summary information are generated “on the side”

Phase 2: Thin Link

- Link only the summary info in a giant index: thin-link
- No need to parse the IR
- Fully-parallel cross-module function importing based on summary.
ThinLTO: Overview

Phase 1: Compile

- Compile
- Extra per-function summary information are generated “on the side”

Phase 2: Thin Link

- Thin Link
- Link only the summary info in a giant index: thin-link.
  - **No need to parse the IR**
- Fully-parallel cross-module function importing based on summary.
- Imported functions are dropped after inlining.

Fully-parallel frontend processing + initial optimizations
ThinLTO: Overview

Phase 1: Compile

- Compile

Phase 2: Thin Link

- Thin Link

Phase 3: Backends

- Backends

Traditional Linking

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.

No need to parse the IR

Fully-parallel cross-module function importing based on summary.
Imported functions are dropped after inlining.

Fully-parallel (very boring) usual optimizations and CodeGen
ThinLTO Model: Summary Generation

```cpp
int test1(int a);
int main() {
    return test1(42);
}
```

```cpp
int test2(int);
int test1(int a) {
    // Do stuff
    return test2(a+1);
}
int test1_foo(int bar) {
    return bar-1;
}
```

```cpp
int test1_foo(int);
int test2(int b) {
    // Do stuff
    return test1_foo(b + 1);
}
```
ThinLTO Model: Summary Generation

```cpp
int test1(int a);
int main() {
  return test1(42);
}
```

```cpp
int test2(int);
int test1(int a) {
  // Do stuff
  return test2(a+1);
}
int test1_foo(int bar) {
  return bar-1;
}
```

```
1010110101010101110101101011
0101010101110101101011010101
0101110101101011010101010111
0101101011010101010111010110
1011010101010111101101011011
1011010101010111101101011011
```
**ThinLTO Model: Summary Generation**

```cpp
int test1(int a);
int main() {
    return test1(42);
}
```

```cpp
int test2(int);
int test1(int a) {
    // Do stuff
    return test2(a + 1);
}
int test1_foo(int b) {
    // Do stuff
    return test1_foo(b + 1);
}
```

```cpp
int test3_foo(int);
```
ThinLTO Model: Summary Generation

```
main.o
```
```
test1.o
```
```
test2.o
```

```
int test1(int a);
int main() {
    return test1(42);
}
```
```
int test2(int);
int test1(int a) {
    // Do stuff
    return test2(a+1);
}
int test1foo(int bar) {
    return bar-1;
}
```
```
int test3(int b) {
    // Do stuff
    return test1foo(b + 1);
}
```

```
clang -cc1 -flto=thin
```
```
source_filename = "main.cpp"
define i32 @main() #0 {
    %1 = call i32 @Ztest11(i32 42)
    ret i32 %1
declare i32 @Ztest11(i32) #1
```
```
source_filename = "test1.cpp"
define i32 @Ztest11(i32) #0 {
    %2 = add nsw i32 %0, 1
    ; Other stuff here...
    %3 = call i32 @Ztest21(i32 %0)
    ret i32 %3
    %4 = call i32 @Ztest11(i32 %0)
    #2 = add nsw i32 %2, 1
    %5 = add nsw i32 %2, -1
    ret i32 %5
declare i32 @Ztest11(i32) #1
```
```
source_filename = "test2.cpp"
define i32 @Ztest21(i32) #0 {
    %2 = add nsw i32 %0, 1
    %3 = call i32 @Ztest11foo(i32 %0)
    ret i32 %3
    %4 = call i32 @Ztest11foo(i32 %0)
    %5 = add nsw i32 %4, 1
    %6 = add nsw i32 %4, -1
    ret i32 %6
declare i32 @Ztest11foo(i32) #1
```
```
Summaries can contain profile data (PGO) data as well, or be extended with other attributes.
Summaries can contain profile data (PGO) as well, or be extended with other attributes.
ThinLTO Model: Summary Generation

main.o

```
1    source_filename = "main.cpp"
2    define 132 @main() #0 {
3        %1 = tail call 132 @Z5test1i(132) 42
4        ret 132 %1
5    }
6    declare 132 @Z5test1i(132) #1
```

Summary:
main, 2 instructions, ...

test1.o

```
1    source_filename = "test1.cpp"
2    define 132 @Z5test1i(132) #0 {
3        %2 = add nsw 132 %0, 1
4        ; Other stuff here...
5        %3 = tail call 132 @Z5test2i(132) %2
6        ret 132 %3
7    }
8    define 132 @Z9test1_fooi(132) #2 {
9        %2 = add nsw 132 %0, -1
10       ret 132 %2
11    }
```

Summary:
_Z5test1i, 42 instructions, ...
_Z9test1_fooi, 2 instructions, ...

test2.o

```
1    source_filename = "test2.cpp"
2    define 132 @Z5test2i(132) #0 {
3        ; Other stuff here...
4        %2 = add nsw 132 %0, 1
5        %3 = tail call 132 @Z9test1_fooi(132) %2
6        ret 132 %3
7    }
8    declare 132 @Z9test1_fooi(132) #1
```

Summary:
_Z5test2i, 13 instructions, ...
**ThinLTO Model: Thin-Link Phase**

**main.o**

```c
source_filename = "main.cpp"

define 132 @main() #0 {
    %1 = tail call 132 @Z5test1l(i32 42)
    ret 132 %1
}

declare 132 @Z5test1l(i32) #1
```

Summary:
main, 2 instructions, ...

**test1.o**

```c
source_filename = "test1.cpp"

define 132 @Z9test1_fooi(i32) #0 {
    %2 = add nsw i32 %0, 1
    ; Other stuff here...
    %3 = tail call 132 @Z9test2l(i32 %0)
    ret 132 %3
}

declare 132 @Z9test2l(i32) #1
```

Summary:
_Z5test1l, 42 instructions, ...
_Z9test1_fooi, 2 instructions, ...

**test2.o**

```c
source_filename = "test2.cpp"

define 132 @Z5test2i(i32) #0 {
    %2 = add nsw i32 %0, 1
    ; Other stuff here...
    %3 = tail call 132 @Z9test1_fooi(i32 %0)
    ret 132 %3
}

declare 132 @Z9test1_fooi(i32) #1
```

Summary:
_Z5test2i, 13 instructions, ...

main.o 2 instructions, ...
_Z5test1l, 42 instructions, ...
_Z9test1_fooi, 2 instructions, ...
_Z5test2i, 13 instructions, ...
ThinLTO Model: Thin-Link Phase

LLVM Linker Plugin

 clang -o test -flto main.o test1.o test2.o
 ld [...] -o test main.o test1.o test2.o

Summary:
main, 2 instructions, ...

Summary:
_Z5test1i, 42 instructions, ...
_Z9test1_foo, 2 instructions

Summary:
_Z5test2i, 13 instructions, ...

main.o
define 132 @main() #0 {
  %1 = tail call 132 @Z5test1i(132 %2)
  ret 132 %1
}
declare 132 @Z5test1i(132) #1

test1.o
define 132 @Z5test1i(132) #2 {
  %3 = add nsw 132 %0, 1
  ; OTHER STUFF هنا...
  %4 = tail call 132 @Z5test2i(132 %3)
  ret 132 %3
}
declare 132 @Z5test2i(132) #3

test2.o
define 132 @Z5test2i(132) #4 {
  %5 = add nsw 132 %0, 1
  %6 = tail call 132 @Z5test1i(132 %5)
  ret 132 %6
}
declare 132 @Z5test1i(132) #5
ThinLTO Model: Thin-Link Phase

```sh
c clang -o test -flto main.o test1.o test2.o
ld [...] -o test main.o test1.o test2.o
```

**LLVM Linker Plugin**

**Summary:**
- `main`, 2 instructions, ...
- `_Z5test1i`, 42 instructions, ...
- `_Z9test1_fooi`, 2 instructions, ...
- `_Z5test2i`, 13 instructions, ...

**Thin-link:**
- Produce the combined index
- Sequential step, but **fast** (<1s for clang)
ThinLTO Model: Thin-Link Phase

```
clang -o test -flto main.o test1.o test2.o
ld [...] -o test main.o test1.o test2.o
```

**LLVM Linker Plugin**

**Combined Index**

**Thin-link:**
- Produce the combined index
- Sequential step, but fast (<1s for clang)
**ThinLTO Model: Thin-Link Phase**

**LLVM Linker Plugin**

```bash
clang -o test -flto main.o test1.o test2.o
ld [...] -o test main.o test1.o test2.o
```

**Thin-link:**
- Produce the combined index
- Sequential step, but **fast** (<1s for clang)
ThinLTO Model: Thin-Link Phase

clang -o test -flto main.o test1.o test2.o

ld [...] -o test main.o test1.o test2.o

LLVM Linker Plugin

Thin-link:
- Produce the combined index
- Sequential step, but fast (<1s for clang)
ThinLTO Model: Thin-Link Phase

clang -o test -flto main.o test1.o test2.o

ld [...] -o test main.o test1.o test2.o

LLVM Linker Plugin

Summary:
main, 2 instructions, ...
_Z5test1i, 42 instructions, ...
_Z9test1_fooi, 2 instructions,

-Thin-link:
- Produce the combined index
- Sequential step, but fast (<1s for clang)
- Original function names are lost!

Combined Index

7c897c21 → main.o, 2 instructions, ...

ThinLTO Model: Thin-Link Phase

clang -o test -flto main.o test1.o test2.o

ld [...] -o test main.o test1.o test2.o

LLVM Linker Plugin

Summary:
main, 2 instructions, ...
_Z5test1i, 42 instructions, ...
_Z9test1_fooi, 2 instructions,

-Thin-link:
- Produce the combined index
- Sequential step, but fast (<1s for clang)
- Original function names are lost!
ThinLTO Model: Thin-Link Phase

LLVM Linker Plugin

Summary:
- main, 2 instructions, ...
- _Z5test1i, 42 instructions, ...
- _Z9test1_fooi, 2 instructions

Summary:
- main.o
- test1.o
- test2.o

Thin-link:
- Produce the combined index
- Sequential step, but **fast** (<1s for clang)
- Original function names are lost!

```
clang -o test -flto main.o test1.o test2.o
```
ThinLTO Model: Thin-Link Phase

```bash
clang -o test -flto main.o test1.o test2.o
ld [...] -o test main.o test1.o test2.o
```

**LLVM Linker Plugin**

**Summary:**
- `main`, 2 instructions, ...
- `_Z5test1i`, 42 instructions, ...
- `_Z9test1_fooi`, 2 instructions, ...
- `_Z5test2i`, 13 instructions, ...

**Thin-link:**
- Produce the combined index
- Sequential step, but **fast** (<1s for clang)
- Original function names are lost!
ThinLTO Model: Thin-Link Phase

```
clang -o test -flto main.o test1.o test2.o
ld [...] -o test main.o test1.o test2.o
```

**Thin-link:**
- Produce the combined index
- Sequential step, but **fast** (<1s for clang)
- Original function names are lost!
ThinLTO *Original* Model: Backend Phase
ThinLTO **Original** Model: Backend Phase

```bash
clang -o test -flto main.o test1.o test2.o
ld [...] -o test main.o test1.o test2.o
```

**LLVM Linker Plugin**

**Combined Index**
- 7c897c21 → main.o, 2 instructions, ...
- 3597eb0 → test1.o, 42 instructions, ...
- 768595e → test1.o, 2 instructions, ...
- 77c4a42 → test2.o, 13 instructions, ...

**Parallel backend:**
- Cross-Module Importing
- Optimization pipeline
- Code generation
Summary:
main, 2 instructions, ...

Summary:
_Z5test1i, 42 instructions, ...
_Z9test1_fooi, 2 instructions,

Summary:
_Z5test2i, 13 instructions, ...

 LLVM Linker Plugin

Combined Index

7c897c21 → main.o, 2 instructions, ...
3597eb0 → test1.o, 42 instructions, ...
768595e → test1.o, 2 instructions, ...
77c4a42 → test2.o, 13 instructions, ...

ThinLTO Original Model: Backend Phase

clang -o test -flto main.o test1.o test2.o
ld [...] -o test main.o test1.o test2.o
**ThinLTO Original Model: Backend Phase**

```
clang -o test -flto main.o test1.o test2.o
ld [
```

```
... -o test main.o test1.o test2.o
```

### LLVM Linker Plugin

**Combined Index**

- 7c897c21 → main.o, 2 instructions, ...
- 3597eb0 → test1.o, 42 instructions, ...
- 768595e → test1.o, 2 instructions, ...
- 77c4a42 → test2.o, 13 instructions, ...

### Listings

**main.o**

```cpp
main.cpp

source_filename = "main.cpp"

define 132 @main() #0 {
  %1 = call i32 @Z5test1i(i32 42)
  ret i32 %1
}

declare i32 @Z5test1i(i32 42) #1
```

**test1.o**

```cpp
main.cpp

source_filename = "main.cpp"

define 132 @Z5test1i(i32 42) #0 {
  %2 = add nsw i32 %0, 1
  %3 = tail call i32 @Z5test2i(i32 %2)
  ret i32 %3
}

declare i32 @Z5test2i(i32) #1
```

**test2.o**

```cpp
main.cpp

source_filename = "main.cpp"

define 132 @Z5test2i(i32) #0 {
  %2 = add nsw i32 %0, 1
  %3 = tail call i32 @Z5test1i(i32 %2)
  ret i32 %3
}

declare i32 @Z5test1i(i32 42) #1
```
**ThinLTO Original Model: Backend Phase**

```
clang -o test -flto main.o test1.o test2.o
```

LLVM Linker Plugin

**Combined Index**

- `7c897c21` ➔ `main.o, 2 instructions, ...`
- `3597eb0` ➔ `test1.o, 42 instructions, ...`
- `768595e` ➔ `test1.o, 2 instructions, ...`
- `77c4a42` ➔ `test2.o, 13 instructions, ...`
**ThinLTO Original Model: Backend Phase**

```
clang -o test -flto main.o test1.o test2.o
ld [...] -o test main.o test1.o test2.o
```

**LLVM Linker Plugin**

---

**main.o**

```cpp
1 #define 132 @main() #0 {
2     %1 = tail call 132 @Z@test1i(132) 42
3     ret 132 %3
4 }
5
declare 132 @Z@test1i(132) #1
```

**Summary:** main, 2 instructions, ...

---

**test1.o**

```cpp
2 #define 132 @Z@test1i(132) #0 {
3     %1 = add new 132 %0, 1
4     %2 = add new 132 %0, 1
5     %3 = add new 132 %0, -1
6     ret 132 %3
7 }
8
define 132 @Z@test1 fooi(132) #0 {
9     %1 = add new 132 %0, -1
10    ret 132 %1
11 }
```

**Summary:** _Z5test1i, 42 instructions, ...

---

**test2.o**

```cpp
2 #define 132 @Z@test2i(132) #0 {
3     %1 = add new 132 %0, 1
4     %2 = add new 132 %0, 1
5     %3 = add new 132 %0, -1
6     ret 132 %3
7 }
8
define 132 @Z@test1 fooi(132) #0 {
9     %1 = add new 132 %0, -1
10    ret 132 %1
11 }
```

**Summary:** _Z5test2i, 13 instructions, ...

---

**Combined Index**

- 7c897c21 → main.o, 2 instructions, ...
- 3597eb0 → test1.o, 42 instructions, ...
- 768595e → test1.o, 2 instructions, ...
- 77c4a42 → test2.o, 13 instructions, ...
LLVM Linker Plugin

```
llvm-link -flto -o test test1.o test2.o
```

Combined Index:
- 7c897c21 → main.o, 2 instructions, ...
- 3597eb0 → test1.o, 42 instructions, ...
- 768595e → test1.o, 2 instructions, ...
- 77c4a42 → test2.o, 13 instructions, ...

Summary:
- main.o, 2 instructions, ...
- test1.o, 42 instructions, ...
- test2.o, 2 instructions, ...

ThinLTO Original Model: Backend Phase
### Combined Index

- 3597eb0 ➔ main.o, 2 instructions, ...
- 3597eb0 ➔ test1.o, 42 instructions, ...
- 768595 ➔ test1.o, 2 instructions, ...
- 77c4a42 ➔ test2.o, 13 instructions, ...

### Main Code Snippet

```cpp
source_filename = "main.cpp"

define 132 @main() #0 {
  %l = tail call 132 @Z5test1i(132) #2
  ret 132 %l
}
declare 132 @Z5test1i(132) #1
```

### Test1 Code Snippet

```cpp
source_filename = "test1.cpp"

define 132 @Z5test1i(132) #0 {
  %l = add msb 132 %l, 1
  %r = tail call 132 @Z5test2i(132) #0
  ret 132 %r
}
declare 132 @Z5test2i(132) #1
```

### Test2 Code Snippet

```cpp
source_filename = "test2.cpp"

define 132 @Z5test2i(132) #0 {
  %l = add msb 132 %l, 1
  %r = tail call 132 @Z9test1_foo(132) #2
  ret 132 %r
}
declare 132 @Z9test1_foo(132) #1
```
**ThinLTO Original Model: Backend Phase**

```
clang -o test -flto main.o test1.o test2.o
```
ThinLTO Original Model: Backend Phase

LLVM Linker Plugin

```
clang -o test -flto main.o test1.o test2.o
ld [...] -o test main.o test1.o test2.o
```
**ThinLTO Original Model: Backend Phase**

```
clang -o test -flto main.o test1.o test2.o
```

**LLVM Linker Plugin**

```
ld [...] -o test main.o test1.o test2.o
```

---

**Summary:**
- `%5 = tail call i32 @Z5test1i(i32) #0` (7c897c21)
- `%6 = add i32 %5, -1` (77c4a42)
- `%7 = tail call i32 @Z5test2i(i32) #1` (768595e)
- `%8 = add i32 %7, -1` (77c4a42)
- `%9 = tail call i32 @Z9test1_fooi(i32) #2` (359eb0)
- `%10 = add i32 %9, -1` (77c4a42)
- `%11 = tail call i32 @Z9test2_fooi(i32) #3` (768595e)
- `%12 = add i32 %11, -1` (77c4a42)

---

**Combined Index**

- `7c897c21 → main.o, 2 instructions, ...
- 359eb0 → test1.o, 42 instructions, ...
- 768595e → test1.o, 2 instructions, ...
- 77c4a42 → test2.o, 13 instructions, ...`
LLVM Linker Plugin

**Summary:**
- main: 2 instructions, ...
- test1: 42 instructions, ...
- test2: 13 instructions, ...

```
clang -o test -flto main.o test1.o test2.o
ld [... ] -o test main.o test1.o test2.o
```
**ThinLTO Original Model: Backend Phase**

```
clang -o test -flto main.o test1.o test2.o
```

**LLVM Linker Plugin**

```
main.o
```

```
define 132 @main() #0 {
  %l = tail call 132 @Z5test11(i132) #2
do ret 132 %1
}
```

Summary: main, 2 instructions, ...

```
test1.o
```

```
define 132 @Z5test11(i132) #2 {
  %1 = tail call 132 @Z5test11(i132) #2
  ret 132 %3
}
```

Summary: _Z5test11, 42 instructions, ...

```
test2.o
```

```
define 132 @Z5test12(i132) #1 {
  %1 = tail call 132 @Z5test12(i132) #1
  ret 132 %3
}
```

Summary: _Z5test2i, 13 instructions, ...

**Combined Index**

- 7c897c21 → main.o, 2 instructions, ...
- 3597eb0 → test1.o, 42 instructions, ...
- 768595e → test1.o, 2 instructions, ...
- 77c4a42 → test2.o, 13 instructions, ...
- 77c4a42
LLVM Linker Plugin

**Summary:**
- main.o: 2 instructions, ...
- test1.o: 42 instructions, ...
- test2.o: 13 instructions, ...

```plaintext
LLVM Linker Plugin

```

**Combined Index**
- 7c897c21 → main.o, 2 instructions, ...
- 3597eb0 → test1.o, 42 instructions, ...
- 768595e → test1.o, 2 instructions, ...
- 77c4a42 → test2.o, 13 instructions, ...

```

**ThinLTO Original Model: Backend Phase**

```shell
clang -o test -flto main.o test1.o test2.o
ld [...] -o test main.o test1.o test2.o
```

```
```
**ThinLTO Original Model: Backend Phase**

```
clang -o test -flto main.o test1.o test2.o
ld [...] -o test main.o test1.o test2.o
```

LLVM Linker Plugin

---

**Combined Index**

- **7c897c21** → `main.o`, 2 instructions, ...
- **3597eb0** → `test1.o`, 42 instructions, ...
- **768595e** → `test1.o`, 2 instructions, ...
- **77c4a42** → `test2.o`, 13 instructions, ...

---

**Source Code**

- **main.o**
  ```cpp
  define int main() { 
    %1 = tail call int @Z5test1l(int2 42) 
    ret int %1 
  }
  ```

- **test1.o**
  ```cpp
  define int @Z5test1l(int2) { 
    %2 = add new int2 0, 1 
    %3 = tail call int @Z5test1l(int2) 
    ret int %3 
  }
  ```

- **test2.o**
  ```cpp
  define int @Z5test2l(int2) { 
    %4 = add new int2 0, 1 
    %5 = tail call int @Z9test1l Foo::int2 132 %2 
    ret int %5 
  }
  ```

---

Summary:

- main, 2 instructions, ...
- _Z5test1l, 42 instructions, ...
- _Z9test1l Foo::int2, 2 instructions,
LLVM Linker Plugin

### main.o
```
source_filename = 'main.cpp'

define i32 @main() #0 (
   %1 = tail call i32 @Ztest1i(i32 42)
   ret i32 42 )

```

Summary: main, 2 instructions, ...

### test1.o
```
source_filename = 'test1.cpp'

define i32 @Ztest1i(i32 42) #0 {
   %2 = add nsw i32 %1, 1
   %3 = tail call i32 @Ztest2i(i32 %2)
   ret i32 %3
}

```

Summary: _Ztest1i, 42 instructions, ...

```
source_filename = 'test2.cpp'

define i32 @Ztest2i(i32 42) #0 {
   ; other stuff here...
   %3 = tail call i32 @Ztest1i(i32 %2)
   ret i32 %3
}

```

Summary: _Ztest2i, 13 instructions, ...

```
define i32 @Ztest1i(i32 42) #0 {
   %1 = tail call i32 @Ztest2i(i32 42)
   ret i32 42
}

```

Summary: main, 2 instructions, ...

### test2.o
```
define i32 @Ztest2i(i32 42) #0 {
   ; other stuff here...
   %3 = tail call i32 @Ztest1i(i32 %2)
   ret i32 %3
}

```

Summary: main, 2 instructions, ...

```
define i32 @Ztest1i(i32 42) #0 {
   %1 = tail call i32 @Ztest2i(i32 42)
   ret i32 42
}

```

Summary: main, 2 instructions, ...

```

define i32 @Ztest2i(i32 42) #0 {
   ; other stuff here...
   %3 = tail call i32 @Ztest1i(i32 %2)
   ret i32 %3
}

```

Summary: main, 2 instructions, ...

### Combined Index
- `7c897c21` → main.o, 2 instructions, ...
- `3597eb0` → test1.o, 42 instructions, ...
- `768595e` → test1.o, 2 instructions, ...
- `77c4a42` → test2.o, 13 instructions, ...

```
clang -o test -flto main.o test1.o test2.o

ld [...] -o test main.o test1.o test2.o

```

ThinLTO Original Model: Backend Phase
LLVM Linker Plugin

ThinLTO Original Model: Backend Phase

```bash
clang -o test -flto main.o test1.o test2.o
```

```
ld [...] -o test main.o test1.o test2.o
```

### Combined Index

- **main.o**, 2 instructions, ...
- **test1.o**, 42 instructions, ...
- **test2.o**, 13 instructions, ...

### Sources

**main.o**

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

**test1.o**

```cpp
#include <iostream>

int test1() {
    std::cout << "Hello, test 1!" << std::endl;
    return 0;
}
```

**test2.o**

```cpp
#include <iostream>

int test2() {
    std::cout << "Hello, test 2!" << std::endl;
    return 0;
}
```
**ThinLTO Original Model: Backend Phase**

```bash
clang -o test -flto main.o test1.o test2.o
```

**LLVM Linker Plugin**

---

**Main File:**

```cpp
source_filename = "main.cpp"
1
2 define 132 @main() #0 {
3 \%1 = call 132 @Ztest1i(132) #2
4 ret 132 \%1
5 }
6
7 declare 132 @Ztest1i(I32) #1
8
9}
```

Summary:

- main, 2 instructions, ...

---

**Test1 File:**

```cpp
source_filename = "test1.cpp"
2 define 132 @Ztest1i(I32) #0 {
3 \%1 = call 132 @Ztest1i(132) #2
4 ret 132 \%1
5 }
7
6 define 132 @Ztest1_foo(132) #2 {
7 \%0 = call 132 @Ztest1(132) #3
8 ret 132 \%0
9 }
10
11}
```

Summary:

- Ztest1i, 42 instructions, ...
- Ztest1_foo, 2 instructions

---

**Test2 File:**

```cpp
source_filename = "test2.cpp"
2 define 132 @Ztest2i(I32) #0 {
3 \%1 = call 132 @Ztest2i(132) #2
4 ret 132 \%1
5 }
7
6 define 132 @Ztest2_foo(132) #2 {
7 \%0 = call 132 @Ztest2(132) #3
8 ret 132 \%0
9 }
10
11}
```

Summary:

- Ztest2i, 13 instructions, ...

---

**Combined Index**

- 7c897c21 → main.o, 2 instructions, ...
- 3597eb0 → test1.o, 42 instructions, ...
- 768595e → test1.o, 2 instructions, ...
- 77c4a42 → test2.o, 13 instructions, ...

---

**Note:**

- The code snippets and index entries are presented for illustration purposes and may not reflect the exact content of the document.
LLVM Linker Plugin

```
clang -o test -flto main.o test1.o test2.o
ld [...] -o test main.o test1.o test2.o
```

**Combined Index**

- 7c897c21 → main.o, 2 instructions, ...
- 3597eb0 → test1.o, 42 instructions, ...
- 768595e → test1.o, 2 instructions, ...
- 77c4a42 → test2.o, 13 instructions, ...

**ThinLTO Original Model: Backend Phase**

- **main.o**
  - `main` function, 2 instructions, ...
- **test1.o**
  - `_Z5test1i` function, 42 instructions, ...
  - `_Z9test1_fooi` function, 2 instructions, ...
- **test2.o**
  - `_Z5test2i` function, 13 instructions, ...

**Source Code Listings**

- **main.c**
  ```
  void main() {
    int a = 1, b = 2;
    int c = a + b;
  }
  ```

- **test1.c**
  ```
  void test1() {
    int a = 3, b = 4;
    int c = a + b;
  }
  ```

- **test2.c**
  ```
  void test2() {
    int a = 5, b = 6;
    int c = a + b;
  }
  ```
**ThinLTO Original Model: Backend Phase**

```bash
clang -o test -flto main.o test1.o test2.o
```

**Combined Index**

- `7c897c21` → `main.o, 2 instructions, ...`
- `3597eb0` → `test1.o, 42 instructions, ...`
- `768595e` → `test1.o, 2 instructions, ...`
- `77c4a42` → `test2.o, 13 instructions, ...`
**ThinLTO Original Model: Backend Phase**

```
clang -o test -flto main.o test1.o test2.o
ld [...] -o test main.o test1.o test2.o
```

**LLVM Linker Plugin**

```c
#define 132 @Ztest1i(132) #0 {
  %1 = tail call 132 @Ztest1i(132 42)
  ret 132 %1
}
declare 132 @Ztest1i(132) #1

#define 132 @Ztest2i(132) #0 {
  %2 = add now 132 %0, 1
  ; Other stuff here...
  %3 = tail call 132 @Ztest2i(132 %2)
  ret 132 %3
}
declare 132 @Ztest2i(132) #1
```

**Combined Index**

- `78c97c21` → main.o, 2 instructions, ...
- `3597eb0` → test1.o, 42 instructions, ...
- `768595e` → test1.o, 2 instructions, ...
- `77c4a42` → test2.o, 13 instructions, ...

**Summary:**
- `main`, 2 instructions, ...
- `test1`, 42 instructions, ...
- `test2`, 2 instructions, ...

**Note:**
- The LLVM Linker Plugin is used to combine the object files into a single executable.
- The `ld` command links the object files together.
- The Combined Index provides a summary of the instructions across the files.
LLVM Linker Plugin

```
clang -o test -flto main.o test1.o test2.o
```

**ThinLTO Original Model: Backend Phase**

```
ld [...o test main.o test1.o test2.o
```

**combined Index**

- 7c897c21 → main.o, 2 instructions, ...
- 3597eb0 → test1.o, 42 instructions, ...
- 768595e → test1.o, 2 instructions, ...
- 77c4a42 → test2.o, 13 instructions, ...

**Test 1 Source**

```
main.o
source_filename = "main.cpp"

1 define 132 @main() {  
2     %1 = call 132 @Z5test1i(132 42)  
3     ret 132 %1 
4 }

... 
```

**Test 2 Source**

```
test1.o
source_filename = "test1.cpp"

1 define 132 @Z5test1i(132) {  
2     %2 = add msb 132 %0, 1  
3     ; Other stuff here...  
4     %3 = call 132 @Z5test2i(132 %0)  
5     ret 132 %3 
6 }

... 
```

**Test 3 Source**

```
test2.o
source_filename = "test2.cpp"

1 define 132 @Z5test2i(132) {  
2     %2 = add msb 132 %0, 1  
3     ; Other stuff here...  
4     %3 = call 132 @Z5test3i(132 %0)  
5     ret 132 %3 
6 }

... 
```
ThinLTO **Original** Model: Backend Phase

```
clang -o test -flto main.o test1.o test2.o
```

LLVM Linker Plugin

```
ld [... ] -o test main.o test1.o test2.o
```

### LLVM Combined Index

- **main.o**
  - 2 instructions
- **test1.o**
  - 42 instructions
- **test2.o**
  - 2 instructions

```cpp
// main.cpp

int main() {
    return 0;
}
```

```cpp
// test1.cpp

int test1() {
    return 0;
}
```

```cpp
// test2.cpp

int test2() {
    return 0;
}
```
ThinLTO

Euro-LLVM 2015: A Fine-Grained Demand-Driven Infrastructure
ThinLTO

Euro-LLVM 2015: A Fine-Grained Demand-Driven Infrastructure

• December 2015: clang bootstrap on MacOS \o/
ThinLTO

Euro-LLVM 2015: A Fine-Grained Demand-Driven Infrastructure

- December 2015: clang bootstrap on MacOS \
  Parallel
ThinLTO

Euro-LLVM 2015: A Fine-Grained Demand-Driven Infrastructure

• December 2015: clang bootstrap on MacOS \o/

  Parallel Incremental
ThinLTO

Euro-LLVM 2015: A Fine-Grained Demand-Driven Infrastructure

• December 2015: clang bootstrap on MacOS \o/
  Parallel  Incremental  Memory lean
ThinLTO

Euro-LLVM 2015: A Fine-Grained Demand-Driven Infrastructure

• December 2015: clang bootstrap on MacOS \o/
  Parallel Incremental Memory lean

• January: link time on single machine was terrible:
  ➔ cross-module importing alone >3x the total time of Monolithic LTO
ThinLTO

Euro-LLVM 2015: A Fine-Grained Demand-Driven Infrastructure

• December 2015: clang bootstrap on MacOS \o/

  Parallel Incremental Memory lean

• January: link time on single machine was terrible:

  ➜ cross-module importing alone >3x the total time of Monolithic LTO

  ➜ linking opt: 1036 inputs files, but >30000 IR loads
ThinLTO

Euro-LLVM 2015: A Fine-Grained Demand-Driven Infrastructure

- December 2015: clang bootstrap on MacOS \
  Parallel Incremental Memory lean

- January: link time on single machine was terrible:
  - cross-module importing alone >3x the total time of Monolithic LTO
  - linking opt: 1036 inputs files, but >30000 IR loads
  - Loading IR is slow! Loading IR with debug-info is insanely slow!
ThinLTO

Euro-LLVM 2015: A Fine-Grained Demand-Driven Infrastructure

- December 2015: clang bootstrap on MacOS \(\odo/\)
  - Parallel
  - Incremental
  - Memory lean
- January: link time on single machine was _terrible_:
  - ➔ cross-module importing alone >3x the total time of Monolithic LTO
  - ➔ linking _opt_: 1036 inputs files, but >30000 IR loads
  - ➔ Loading IR is slow! Loading IR with debug-info is _insanely_ slow!

“none of the subsystems in LLVM are really good until they have been rewritten at least once.”

 ➔ This is what we did between January and March!
ThinLTO

Euro-LLVM 2015: A Fine-Grained Demand-Driven Infrastructure

• December 2015: clang bootstrap on MacOS \o/
  Parallel Incremental Memory lean

• January: link time on single machine was terrible:
  ➔ cross-module importing alone >3x the total time of Monolithic LTO
  ➔ linking opt: 1036 inputs files, but >30000 IR loads
  ➔ Loading IR is slow! Loading IR with debug-info is insanely slow!

“none of the subsystems in LLVM are really good until they have been rewritten at least once.”

 ➔ This is what we did between January and March!
ThinLTO: Overview

Phase 1: Compile

Phase 2: Thin Link

Phase 3: Backends

Traditional Linking
ThinLTO: Overview

Phase 1: Compile
- Frontend
- Linker

Phase 2: Thin Link

Phase 3: Backends
- Opt
- BE

Traditional Linking

+ Local reference/call graph
+ Module Hash (for incremental build)
ThinLTO: Overview

Phase 1: Compile

Frontend

Phase 2: Thin Link

Phase 3: Backends


BE BE BE BE BE BE BE BE

+ Local reference/call graph
+ Module Hash (for incremental build)

Traditional Linking
ThinLTO: Overview

Phase 1: Compile
- Frontend
- Linker

Phase 2: Thin Link

Phase 3: Backends

Traditional Linking

+ Local reference/call graph
+ Module Hash (for incremental build)
ThinLTO: Overview

Phase 1: Compile
- Frontend
- Linker

Phase 2: Thin Link
- .bc
- .bc
- .bc
- .bc
- .bc
- .bc
- .bc
- .bc

Phase 3: Backends
- Opt
- Opt
- Opt
- Opt
- Opt
- Opt
- Opt
- Opt

Traditional Linking

- Local reference/call graph
- Module Hash (for incremental build)
- Full reference/call graph
ThinLTO: Overview

Phase 1: Compile
- Frontend
- Linker

Phase 2: Thin Link
- Inter-procedural Analysis
  - + Local reference/call graph
  - + Module Hash (for incremental build)
  - + Full reference/call graph

Phase 3: Backends
- Traditional Linking
  - Opt
  - BE
ThinLTO: Overview

Phase 1: Compile
- .bc
- .bc
- .bc
- .bc
- .bc
- .bc
- .bc
- .bc
- .bc

Phase 2: Thin Link
- Inter-procedural Analysis
- Analyses Results

Phase 3: Backends
- Opt
- Opt
- Opt
- Opt
- Opt
- Opt
- Opt
- Opt
- BE
- BE
- BE
- BE
- BE
- BE
- BE
- BE

+ Local reference/call graph
+ Module Hash (for incremental build)
+ Full reference/call graph

Traditional Linking
ThinLTO: Overview

Phase 1: Compile

- Frontend
  - .bc

- Linker
  - .bc

Phase 2: Thin Link

- Inter-procedural Analysis
- Analyses Results

Phase 3: Backends

- Opt
- Opt
- Opt
- Opt
- Opt
- Opt
- Opt
- Opt

Traditional Linking

- Local reference/call graph
- Module Hash (for incremental build)
- Full reference/call graph
- Parallel Inter-procedural transformation based on the analyses results
ThinLTO Revisited: Compile Phase

`main.o`

Summary:
- `main`, 2 instructions

`t1.o`

Summary:
- `_Z5test1i`, 42 instructions
- `_Z9test1_fooi`, 2 instructions

`t2.o`

Summary:
- `_Z5test2i`, 13 instructions

```
cpp -cc1 -flto=thin
```
ThinLTO Revisited: Compile Phase

**Summary:**
- **main:** 2 instructions
- **_Z5test1i:** 42 instructions
- **_Z9test1_fooi:** 2 instructions
- **_Z5test2i:** 13 instructions

**clang -cc1 -flto=thin**

- **main.o**: `int test1(int a);` 
  - Global linkage, call `_Z5test1i`

- **test1.o**: 
  - `int test1(int a) {` 
  - `return test1(42);` 
  - `}` 
  - `int main() {` 
  - `return test1(42);` 
  - `}` 
  - `}` 
  - Define `132` @main() #0 ( 
  - `v1 = tail call 132 @Z5test1i(132 42)` 
  - `ret 132 v1` 
  - `declare 132 @Z5test1i(132) #1` 

- **test2.o**: 
  - `int test2_foo(int);` 
  - `int test2(int b) {` 
  - `// Do stuff` 
  - `// ...` 
  - `return test2(a+1);` 
  - `}` 
  - `int test1_foo(int bar) {` 
  - `return Bar-1;` 
  - `}` 

- **source_filename = "main.cpp"**
- **source_filename = "test1.cpp"**
- **source_filename = "test2.cpp"**
ThinLTO Revisited: Compile Phase

```
main.cpp
1 int test1(int a);
2 int main() {
3   return test1(42);
4 }
```

```
test1.cpp
1 int test2(int);
2 int test1(int a) {
3   // Do stuff
4   // ...
5   return test2(a+1);
6 }
7 int test1_foo(int bar) {
8   return Bar-1;
9 }
```

```
test2.cpp
1 int test1_foo(int);
2 int test2(int b) {
3   // Do stuff
4   return test1_foo(b + 1);
5 }
```

```
clang -cc1 -flto=thin
```

```
main.o
1 source_filename = "main.cpp"
2 define 132 @main() #0 {
3   %1 = tail call 132 @Z5test1(132 42)
4   ret 132 %1
5 }
6 declare 132 @Z5test1(132) #1
```

Summary:
main, 2 instructions
global linkage, call _Z5test1

```
test1.o
1 source_filename = "test1.cpp"
2 define 132 @test1_foo(132) #0 {
3   %2 = add new 132 %0, 1
4   %3 = tail call 132 @Z2test1_foo(132 %2)
5   ret 132 %3
6 }
7 declare 132 @Z2test1_foo(132) #1
```

Summary:
_Z5test1, 42 instructions
global linkage, call _Z5test1
_Z9test1_foo, 2 instructions
global linkage, call _Z9test1_foo

```
test2.o
1 source_filename = "test2.cpp"
2 define 132 @test1_foo(132) #0 {
3   %2 = add new 132 %0, 1
4   %3 = tail call 132 @Z9test1_foo(132 %2)
5   ret 132 %3
6 }
7 declare 132 @Z9test1_foo(132) #1
```

Summary:
_Z5test2i, 13 instructions
global linkage, call _Z9test1_foo
ThinLTO Revisited: Compile Phase

**main.o**

```c
source_filename = "main.cpp"

define 132 @main() #0 {
    %1 = tail call 132 @Z5test1i(132 42)
    ret 132 %1
}

declare 132 @Z5test1i(132) #1
```

Summary:

- main, 2 instructions
- global linkage, call _Z5test1i

**test1.o**

```c
source_filename = "test1.cpp"

define 132 @Z5test1i(132) #0 {
    %2 = add nsw 132 %0, 1
    ; Other stuff here...
    %3 = tail call 132 @Z5test2i(132 %2)
    ret 132 %3
}

define 132 @Z9test1_fooi(132 %2) #2 {
    %4 = add nsw 132 %0, -1
    %5 = tail call 132 @Z9test1_fooi(132 %2)
    ret 132 %2
}

declare 132 @Z9test1_fooi(132) #1
```

Summary:

- _Z5test1i, 42 instructions
- _Z9test1_fooi, 2 instructions
- global linkage, call _Z5test1i
- global linkage, call _Z9test1_fooi

**test2.o**

```c
source_filename = "test2.cpp"

define 132 @Z5test2i(132) #0 {
    ; Other stuff here...
    %2 = add nsw 132 %0, 1
    %3 = tail call 132 @Z9test1_fooi(132 %2)
    ret 132 %3
}

declare 132 @Z9test1_fooi(132) #1
```

Summary:

- _Z5test2i, 13 instructions
- _Z9test1_fooi, 2 instructions
- global linkage, call _Z5test1_fooi
ThinLTO Revisited: Thin Link

### Summary:
- **main.o**: 2 instructions
  - Global linkage, call `_Z5test1i`
- **test1.o**: 42 instructions
  - Global linkage, call `_Z5test1i`
  - Global linkage, call `_Z9test1_fooi`
- **test2.o**: 13 instructions
  - Global linkage, call `_Z5test2i`
  - Global linkage, call `_Z9test1_fooi`
ThinLTO Revisited: Thin Link

**Thin-link:**
- Produce the combined index
- Sequential step, but fast (<2s for clang)
- Original function names are lost!
Thin-link:
- Produce the combined index
- Sequential step, but fast (<2s for clang)
- Original function names are lost!
**ThinLTO Revisited: Thin Link**

**Thin-link:**
- Produce the combined index
- Sequential step, but fast (<2s for clang)
- Original function names are lost!
Thin-link:

- Produce the combined index
- Sequential step, but fast (<2s for clang)
- Original function names are lost!
- Run inter-procedural analyses:
  - Transformations: Global DCE, Linkage, …
**ThinLTO Revisited: Thin Link**

**Thin-link:**
- Produce the combined index
- Sequential step, but fast (<2s for clang)
- Original function names are lost!
- Run inter-procedural analyses:
  - Transformations: Global DCE, Linkage, …

No need to load any IR! This is still a very fast serial phase.
LLVM Linker Plugin

ThinLTO Revisited: Thin Link

main.o, 2 instructions

Call graph:
- main.o calls test1.o
- test1.o calls test2.o

Combined Index
- 7c897c21 → main.o, 2 instructions
- call 3597eb0
- 3597eb0 → test1.o, 42 instructions
- call 77c4a42
- 768595e → test1.o, 2 instructions
- call 77c4a42
- 77c4a42 → test2.o, 13 instructions
- call 768595e

Global linkage:
- call _Z5test1i
- call _Z5test2i
- call _Z9test1_fooi
ThinLTO Revisited: Thin Link

Import Lists

Combined Index

- 7c897c21 → main.o, 2 instructions
- call 3597eb0
- 3597eb0 → test1.o, 42 instructions
- call 77c4a42
- 768595e → test1.o, 2 instructions
- call 77c4a42
- 77c4a42 → test2.o, 13 instructions
- call 768595e

Summary:

- main, 2 instructions
global linkage, call _Z5test1i

- _Z5test1i, 42 instructions
global linkage, call _Z5test2i
Z9test1_fooi, 2 instructions

- _Z5test2i, 13 instructions
global linkage, call _Z9test1_fooi
ThinLTO Revisited: Thin Link

LLVM Linker Plugin

Summary:
- main.o, 2 instructions
- \_Z5test1\_i, 42 instructions
- \_Z9test1\_foo\_i, 2 instructions
- \_Z5test2\_i, 13 instructions

Global linkage, call _Z5test1\_i
Global linkage, call _Z5test2\_i
Global linkage, call _Z9test1\_foo\_i

Import Lists
- main.o
- 3597eb0
- 77c4a42
- 768595e

Combined Index
- 7c897c21 → main.o, 2 instructions
- call 3597eb0
- 3597eb0 → test1.o, 42 instructions
- call 77c4a42
- 768595e → test1.o, 2 instructions
- call 77c4a42
- 77c4a42 → test2.o, 13 instructions
- call 768595e
ThinLTO Revisited: Thin Link

**LLVM Linker Plugin**

**Import Lists**
- **main.o**
  - call 7c897c21
  - 2 instructions

**test1.o**
- call 3597eb0
- call 77c4a42
- call 768595e
- 42 instructions

**test2.o**
- call 3597eb0
- call 77c4a42
- call 768595e
- 13 instructions

---

**Combined Index**
- 7c897c21 → main.o, 2 instructions
- 3597eb0 → test1.o, 42 instructions
- 77c4a42 → test1.o, 2 instructions
- 768595e → test2.o, 13 instructions
ThinLTO Revisited: Thin Link

**Import Lists**
- **main.o**
  - 3597eb0
  - 77c4a42
  - 768595e
- **test1.o**
  - test1.o
  - test2.o
- **test2.o**
  - test2.o

**Combined Index**
- 7c897c21 → main.o, 2 instructions
- call 3597eb0 → test1.o, 42 instructions
- call 77c4a42 → test2.o, 13 instructions
- 768595e → test1.o, 2 instructions
- call 768595e → test2.o, 13 instructions

**Summary:**
- **main.o**: 2 instructions
- **_Z5test1i**: 42 instructions
- **_Z9test1_fooi**: 2 instructions
- **_Z5test2i**: 13 instructions

**LLVM Linker Plugin**
### Import Lists

- **main.o**
  - 3597eb0
  - 77c4a42
  - 768595e
- **test1.o**
  - 77c4a42
- **test2.o**
  - 768595e

### Combined Index

- **7c897c21** → **main.o**, 2 instructions
- **call 3597eb0**
- **3597eb0** → **test1.o**, 42 instructions
- **call 77c4a42**
- **768595e** → **test1.o**, 2 instructions
- **77c4a42** → **test2.o**, 13 instructions
- **call 768595e**

### Summary

- **main.o**
  - 2 instructions
- **_Z5test1i**
  - 42 instructions
- **_Z9test1_fooi**
  - 2 instructions
- **_Z5test2i**
  - 13 instructions
- **global linkage, call _Z5test1i**
- **global linkage, call _Z5test2i**
- **global linkage, call _Z9test1_fooi**
LLVM Linker Plugin

ThinLTO Revisited: Thin Link

Summary:
main, 2 instructions
global linkage, call _Z5test1i

Summary:
_Z5test1i, 42 instructions
global linkage, call _Z5test2i
_Z9test1_fooi, 2 instructions

Summary:
_Z5test2i, 13 instructions
global linkage, call _Z9test1_fooi

Import Lists

main.o
3597eb0
77c4a42
768595e
test1.o
77c4a42
test2.o
768595e
test1.o
test2.o

ThinLTO Revisited: Parallel Backends

LLVM Linker Plugin

Summary:
main.o, 2 instructions
global linkage, call _Z5test1i

Summary:
_Z5test1i, 42 instructions
global linkage, call _Z5test2i
_Z9test1_fooi, 2 instructions

Summary:
_Z5test2i, 13 instructions
global linkage, call _Z9test1_fooi

Import Lists

main.o
3597eb0
77c4a42
768595e
test1.o
test2.o

test1.o
77c4a42
768595e
test2.o
test1.o
Parallel ThinLTO Backends:

- Read the import lists
**ThinLTO Revisited: Parallel Backends**

**Parallel ThinLTO Backends:**

- Read the import lists
- Process all imports from one module after the other

---

**Import Lists**

- **main.o**
  - 3597eb0
  - 77c4a42
  - 768595e
- **test1.o**
  - 77c4a42
- **test2.o**
  - 768595e
Parallel ThinLTO Backends:

- Read the import lists
- Process all imports from one module after the other

Knowing upfront what to import allows to limit the number of IR loading
ThinLTO Revisited: Parallel Backends

**Summary:**
- **main.o**: 2 instructions
- **test1.o**: 42 instructions
- **test2.o**: 13 instructions

- **global linkage, call _Z5test1i**
- **global linkage, call _Z5test2i**
- **global linkage, call _Z9test1_fooi**

---

```
source_filename = "main.cpp"

define i32 @main() #0 {  
  %1 = tail call i32 @Z5test1i(i32 42)  
  ret i32 %1  
}

define available_externally i32 @Z5test2i(i32) #0 {  
  %2 = add nsw i32 %0, 1  
  ; Other stuff here...  
  %3 = tail call i32 @Z5test2i(i32 %2)  
  ret i32 %3  
}

define available_externally i32 @Z9test1_fooi(i32) #0 {  
  ; Other stuff here...  
  %2 = add nsw i32 %0, 1  
  %3 = tail call i32 @Z9test1_fooi(i32 %2)  
  ret i32 %3  
}
```

---

LLVM Linker Plugin
ThinLTO Revisited: Parallel Backends

LLVM Linker Plugin

main.o

Summary: main, 2 instructions
global linkage, call _Z5test1i

CGSCC Simplification and Inlining

```c
source_filename = "main.cpp"

#define 132 @main() #0 {
  %1 = tail call i32 @Z5test1i(i32 42)
  ret i32 %1
}
```

```c
#define available_externally i32 @Z5test1i(i32) #0 {
  %2 = add nsw i32 %0, 1
  ; Other stuff here...
  %3 = tail call i32 @Z5test2i(i32 %2)
  ret i32 %3
}
```

```c
#define available_externally i32 @Z9test1_fooi(i32) #0 {
  %2 = add nsw i32 %0, 1
  %3 = tail call i32 @Z9test1_fooi(i32 %2)
  ret i32 %3
}
```

```c
#define available_externally i32 @Z9test1_fooi(i32) #2 {
  %2 = add nsw i32 %0, -1
  ret i32 %2
}
```
CGSCC Simplification and Inlining

1) “Simplification” passes: 47 passes including SROA - JumpThreading - InstCombine - LICM - … (and many others*)
ThinLTO Revisited: Parallel Backends

LLVM Linker Plugin

1) “Simplification” passes: 47 passes including SROA - JumpThreading - InstCombine - LICM - … (and many others*)

2) Inlining

1) “Simplification” passes: 47 passes including SROA - JumpThreading - InstCombine - LICM - … (and many others*)

CGSCC Simplification and Inlining

LLVM Linker Plugin
ThinLTO Revisited: Parallel Backends

1) "Simplification" passes: 47 passes including SROA - JumpThreading - InstCombine - LICM - … (and many others*)

2) Inlining

3) "Simplification" passes

CGSCC Simplification and Inlining

*See PassManagerBuilder::addFunctionSimplificationPasses()
**ThinLTO Revisited: Parallel Backends**

1) “Simplification” passes: 47 passes including SROA - JumpThreading - InstCombine - LICM - … (and many others*)

2) Inlining

3) “Simplification” passes

4) Inlining

**CGSCC Simplification and Inlining**

1) “Simplification” passes: 47 passes including SROA - JumpThreading - InstCombine - LICM - … (and many others*)

2) Inlining

3) “Simplification” passes

4) Inlining

*See `PassManagerBuilder::addFunctionSimplificationPasses()`
ThinLTO Revisited: Parallel Backends

CGSCC Simplification and Inlining

1) “Simplification” passes: 47 passes including SROA - JumpThreading - InstCombine - LICM - … (and many others*)

2) Inlining

3) “Simplification” passes

4) Inlining

5) “Simplification” passes

*See PassManagerBuilder::addFunctionSimplificationPasses()
**ThinLTO Revisited: Parallel Backends**

**CGSCC Simplification and Inlining**

6) **Inlining**

4) **Inlining**

5) **“Simplification” passes**

2) **Inlining**

3) **“Simplification” passes**

1) **“Simplification” passes**: 47 passes including SROA - JumpThreading - InstCombine - LICM - … (and many others*)

*See PassManagerBuilder::addFunctionSimplificationPasses()
**ThinLTO Revisited: Parallel Backends**

**LLVM Linker Plugin**

1) **Simplification** passes: 47 passes including SROA - JumpThreading - InstCombine - LICM - … (and many others*)

2) **Inlining**

3) **Simplification** passes

4) **Inlining**

5) **Simplification** passes

6) **Inlining**

7) **“Simplification”** passes

**CGSCC Simplification and Inlining**

1) **“Simplification”** passes: 47 passes including SROA - JumpThreading - InstCombine - LICM - … (and many others*)

2) **Inlining**

3) **“Simplification”** passes

4) **Inlining**

5) **“Simplification”** passes

6) **Inlining**

7) **“Simplification”** passes

*See PassManagerBuilder::addFunctionSimplificationPasses()
ThinLTO Revisited: Parallel Backends

6) Inlining
7) “Simplification” passes

8) “Eliminate Available Externally” Pass

1) “Simplification” passes: 47 passes including SROA - JumpThreading - InstCombine - LICM - … (and many others*)
CGSCC Simplification and Inlining

1) “Simplification” passes: 47 passes including SROA - JumpThreading - InstCombine - LICM - … (and many others*)

2) Inlining

3) “Simplification” passes

4) Inlining

5) “Simplification” passes

6) Inlining

7) “Simplification” passes

8) “Eliminate Available Externally” Pass

9) “Optimization Passes”: 43 passes including vectorization

*See PassManagerBuilder::addFunctionSimplificationPasses()
**ThinLTO Revisited: Parallel Backends**

**LLVM Linker Plugin**

1) "Simplification" passes: 47 passes including SROA - JumpThreading - InstCombine - LICM - … (and many others*)

2) Inlining

3) "Simplification" passes

4) Inlining

5) "Simplification" passes

6) Inlining

7) "Simplification" passes

8) “Eliminate Available Externally” Pass

9) “Optimization Passes”: 43 passes including vectorization

10) Codegen

**Summary:**
- **main.o**
  - 2 instructions
  - Global linkage, call _Z5test1i

- **test1.o**
  - 42 instructions
  - Global linkage, call _Z5test2i

- **test2.o**
  - 13 instructions
  - Global linkage, call _Z9test1_fooi

*See `PassManagerBuilder::addFunctionSimplificationPasses()`
LLVM Linker Plugin

ThinLTO Revisited: Parallel Backends

Summary:
- main, 2 instructions
  - global linkage, call _Z5test1i

Summary:
- _Z5test1i, 42 instructions
  - global linkage, call _Z5test2i
  - Z9test1_fooi, 2 instructions

Summary:
- _Z5test2i, 13 instructions
  - global linkage, call _Z9test1_fooi

source_filename = "test1.cop"
define i32 @Z5test1i(i32) #0 {
  %2 = add nsw i32 %0, 1
  ; Other stuff here...
  %3 = tail call i32 @Z5test2i(i32 %2)
  ret i32 %3
}
define i32 @Z9test1_fooi(i32) #2 {
  %2 = add nsw i32 %0, -1
  ret i32 %2
}
define available_externally i32 @Z5test2i(i32) #0 {
  ; Other stuff here...
  %2 = add nsw i32 %0, 1
  %3 = tail call i32 @Z9test1_fooi(i32 %2)
  ret i32 %3
}
LLVM Linker Plugin

ThinLTO Revisited: Parallel Backends

Summary:
main, 2 instructions
global linkage, call _Z5test1i

Summary:
_Z5test1i, 42 instructions
global linkage, call _Z5test2i
Z9test1_fooi, 2 instructions

Summary:
_Z5test2i, 13 instructions
global linkage, call _Z9test1_fooi

CGSCC Simplification and Inlining

4) Inlining

5) “Simplification” passes

1) “Simplification” passes

2) Inlining

3) “Simplification” passes

6) “Eliminate AvailableExternally” Pass

7) “Optimization Passes”: 43 passes including vectorization
CGSCC Simplification and Inlining

1) “Simplification” passes
2) Inlining
3) “Simplification” passes
4) Inlining
5) “Simplification” passes

“Eliminate Available Externally” Pass
6) “Eliminate Available Externally” Pass
7) “Optimization Passes”: 43 passes including vectorization
8) Codegen

Parallelism is obtained thanks to redundant optimizations over the same IR!
Thin-Link IPA: Correctness
Global Scope Promotion

```cpp
// Test 1
int test2(int x);
int test1(int a) {
    // Do stuff
    ...
    return test2(a+1);
}
int test1_foo(int bar) {
    return bar - 1;
}

// Test 2
int test2(int b) {
    // Do stuff
    return test1_foo(b - 1);
}
```
Thin-Link IPA: Correctness
Global Scope Promotion

```cpp
1: int test2(int a) {
2:   int test1(int b) {
3:     // Do stuff
4:     // ...
5:     return test2(a+1);
6:   }
7:   int test1(int b) {
8:     return Bar-1;
9: }
```
Thin-Link IPA: Correctness

Global Scope Promotion

```
int test2(int);
int test1(int a) {
    // Do stuff
    //...
    return test2(a+1);
}
int test1_foo(int bar) {
    return Bar-1;
}
```

```
int test1_foo(int bar) {
    return Bar-1;
}
```
Thin-Link IPA: Correctness
Global Scope Promotion

```cpp
int test2(int);
int test1(int a) {
  // Do stuff
  // ...
  return test2(a + 1);
}
int test1_foo(int bar) {
  return bar - 1;
}

static void bar() {
  // Important stuff here...
}
int test2(int b) {
  // Do stuff
  bar();
  return test1_foo(b - 1);
}
```
Thin-Link IPA: Correctness
Global Scope Promotion
Introduction of an external reference (due to importing)!
Introduction of an external reference (due to importing)!

→ requires promotion/rename of local linkage functions
Introduction of an external reference (due to importing)!

- requires promotion/rename of local linkage functions
- similar technique for exported “discardable” linkage references
Thin-Link IPA: Compile Time Optimization
Weak Linkage Resolution

```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);
}
```

```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);
}
```
Thin-Link IPA: Compile Time Optimization
Weak Linkage Resolution

C++ template generates a lot of redundant code!
C++ template generates a lot of redundant code!

- Monolithic LTO will *merge* these and codegen only one naturally
Thin-Link IPA: Compile Time Optimization
Weak Linkage Resolution

C++ template generates a lot of redundant code!

- Monolithic LTO will *merge* these and codegen only one naturally
- ThinLTO selects one at Thin Link time ➔ other copies marked for drop after inlining
C++ template generates a lot of redundant code!

- Monolithic LTO will *merge* these and codegen only one naturally
- ThinLTO selects one at Thin Link time 🔄 other copies marked for drop after inlining

Linking clang: ~25% less function that are codegen!
Thin Link IPA: Dead Global Pruning

```cpp
textred int Option = 42;

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

```cpp
textred extern int Option;

void setOption(int Value) {
    Option = Value;
}
```
Thin Link IPA: Dead Global Pruning

```c
int Option = 42;

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

```c
extern int Option;

void setOption(int Value) {
    Option = Value;
}
```
Thin Link IPA: Dead Global Pruning

```c
1 int Option = 42;
2
3 int getGlobalOption() {
4     return Option;
5 }
```

```c
1 extern int Option;
2
3 void setOption(int Value) {
4     Option = Value;
5 }
```
Thin Link IPA: Dead Global Pruning

• Linker identifies external reference to getGlobalOption()
Thin Link IPA: Dead Global Pruning

- Linker identifies external reference to `getGlobalOption()`
- Compute reachability to externally referenced nodes in index
Thin Link IPA: Dead Global Pruning

- Linker identifies external reference to getGlobalOption()
- Compute reachability to externally referenced nodes in index
- Prune unreachable nodes from the graph
**Thin Link IPA: Dead Global Pruning**

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

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

```c
int Option = 42;
int getGlobalOption() {
    return Option;
}

extern int Option;
void setOption(int Value) {
    Option = Value;
}
```
Thin-Link IPA Future Optimization Example: Global Variables

```c
#include <stdio.h>

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

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

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

```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();  // this call is dead
        data = data + 42;
    }
    return data;
}
```
Thin-Link IPA Future Optimization Example: Global Variables

```c
#include <stdio.h>

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

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

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

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

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;
  }
}
```

```c
#include <stdio.h>

extern int i.llvm.A570184;

int main() {
  if (i.llvm.A570184 < 0)
    return printf("Hi\n");
  return 42;
}
```

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

int foo3.llvm.A570184 = 0;

int foo3.llvm.A570184() {
  printf("Hi\n");
  return 10;
}

int foo1(void) {
  if (i.llvm.A570184 < 0)
    return printf("Hi\n");
  return 42;
}
```
Knowing the range for i.llvm.A570184 (here it is even easier: it is a constant) would enable folding the test as false.
Knowing the range for `i.llvm.A570184` (here it is even easier: it is a constant) would enable folding the test as false.

All this code can be dead-striped by the linker, but takes time to optimize/codegen.
All this code can be dead-stripped by the linker, but takes time to optimize/codegen.

Knowing the range for `i.llvm.A570184` (here it is even easier: it is a constant) would enable folding the test as false.

This is just one example of opportunity for better summary-based optimization!
Profile Guided Optimization (PGO): Indirect Call Promotion

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

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

int foo() {
    // do stuff
    // ...
    return bar();
}
```
Profile Guided Optimization (PGO): Indirect Call Promotion

```c++
extern int (*bar)();

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

int foo() {
    // do stuff
    // ...;
    if (bar == &bar_impl)
        return bar_impl();
    return bar();
}
```
Profile Guided Optimization (PGO): Indirect Call Promotion

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

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

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

Indirect Call Promotion can only promote if the target is in the same module.
Profile Guided Optimization (PGO): Indirect Call Promotion

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

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

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

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

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

int (*bar)() = &bar_impl;
```
PGO Indirect Call Promotion

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

int (*bar)() = &bar_impl;
```

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

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

    return bar();
}
```
PGO Indirect Call Promotion

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

int (*bar)() = &bar_impl;
```

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

int foo() {
    // do stuff
    // ...
    return bar();
}
```
PGO Indirect Call Promotion

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

int (*bar)() = &bar_impl;
```

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

int foo() {
    // do stuff
    // ...
    return bar();
}
```
PGO Indirect Call Promotion

**Summary:**

- Function: `_Z5test2i`
- 13 instructions
- Global linkage

```
clang -cc1 -flto=thin
```

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

int (*bar)() = &bar_impl;
```

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

int foo() {
    // do stuff
    // ...
    return bar();
}
```

```
source_filename = "indirect_call.cpp"
@bar = external global i32 (*)*, align 8
define i32 @Zfoo() @0 {
    %1 = load i32 (*)*, i32 (*)* %bar, align 8
    %2 = call i32 @1(@1)()
    ret i32 %2
}
```

Summary:

- `_Z5test2i`, 13 instructions,
- Global linkage,
PGO Indirect Call Promotion

Summary:
_Z5test2i, 13 instructions, global linkage,
The summary records indirect calls possible targets as regular calls (speculative).
LLVM Linker Plugin

PGO Indirect Call Promotion

main.o

```c
source_filename = "main.cpp"

#define main() { 
  // Other stuff here...
  ret __attribute__((noinline))
}

declare __attribute__((noinline))
```

Summary:
main, 2 instructions, ...

```
```

 test1.o

```c
source_filename = "test1.cpp"

#define test1() 
  // Other stuff here...
  ret __attribute__((noinline))
```

declare __attribute__((noinline))

Summary:
_Z5test1i, 42 instructions, ...
_Z9test1_fooi, 2 instructions, ...

```
```

test2.o

```c
source_filename = "test2.cpp"

#define test2() 
  // Other stuff here...
  ret __attribute__((noinline))
```

declare __attribute__((noinline))

Summary:
_Z5test2i, 13 instructions, global linkage, call 768595e

```
```

Thin-link:
Thin-link:

LLVM Linker Plugin

PGO Indirect Call Promotion

main.o

1: source_filename = "main.cpp"
2: define i32 @main() #0 {  
3: %1 = call i32 @Z5test1i(i32 42)  
4: ret i32 %1
5: }
6: declare i32 @Z5test1i(i32) #1

Summary:  
main, 2 instructions, ...

test1.o

1: source_filename = "test1.cpp"
2: define i32 @Z5test1i(i32) #0 {  
3: %2 = add nsw i32 %0, 1  
4: %3 = call i32 @Z5test2i(i32 %2)  
5: ret i32 %3
6: }
7:   declare i32 @Z5test1i(i32) #1

Summary:  
_Z5test1i, 42 instructions, ...

_Z9test1_fooi, 2 instructions, ...

test2.o

1: source_filename = "test2.cpp"
2: define i32 @Z5test2i(i32 %0) #0 @prof #8 {  
3: %0 = load i32 (i32)*, i32 (i32)* @prof, align 8  
4: %add = add nsw i32 %0, 1  
5: %call = call i32 %add(i32 %add), !prof #8  
6: ret i32 %call
7: }
8:   declare i32 @Z5test2i(i32) #1
9:
10:  i32 = 1(1"0", i32 0, 1, 184 0, 184 279134, 184 13

Summary:  
_Z5test2i, 13 instructions,  
global linkage, call 768595e

**Thin-link:**

- Produce the combined index as usual
**Thin-link:**

- Produce the combined index as usual
- Indirect profile-based edge points to target, if in index, the same way as direct edges.
Thin-link:

- Produce the combined index as usual
- Indirect profile-based edge points to target, if in index, the same way as direct edges.
- The indirect call target can be selected for import, just like targets of direct calls.
**Thin-link:**

- Produce the combined index as usual
- Indirect profile-based edge points to target, if in index, the same way as direct edges.
- The indirect call target can be selected for import, just like targets of direct calls.
- Indirect calls are promoted after importing, but before the inliner.
Profile Guided Optimization (PGO): Importing Heuristics

Summary:
- `_Z3hotv`, 150 instructions, global linkage
- `_Z4coldv`, 50 instructions, global linkage
- main, 10 instructions, global linkage,
  - call f9f84d51
  - call f46f05f7

Code excerpts:
```cpp
void hot();
void cold();
void foo(bool TakeHotPath) {
    if (TakeHotPath) {
        hot();
    } else {
        cold();
    }
}

void hot() {
    // do stuff...
    // 150 instructions here
}

void cold() {
    // do stuff...
    // 50 instructions here
}
```
Profile Guided Optimization (PGO): Importing Heuristics

Summary:
_main, 10 instructions, global linkage,
call f9f84d51
call f46f05f7

_Z3hotv, 150 instructions, global linkage
_Z4coldv, 50 instructions, global linkage

⇒ Only _cold() would be inlined!
Profile Guided Optimization (PGO): Importing Heuristics

Summary:
- main, 10 instructions, global linkage,
  - call f9f84d51
  - call f46f05f7

→ Only cold() would be inlined!

The importer would only import cold()!

Summary:
- _Z3hotv, 150 instructions, global linkage
- _Z4coldv, 50 instructions, global linkage
Profile Guided Optimization (PGO): Importing Heuristics

Profile:

- **main**, 10 instructions, global linkage,
  - call f9f84d51
  - call f46f05f7

- **_Z3hotv**, 150 instructions, global linkage
- **_Z4coldv**, 50 instructions, global linkage

Summary:

- **cold()** would not be inlined!
- **hot()** would be inlined if available

With PGO data, **cold** will not be inlined, **hot** would be inlined if available

The importer would only import **cold()**!
Profile Guided Optimization (PGO): Importing Heuristics

Only `cold()` would be inlined!

The importer would only import `cold()`!

With PGO data, `cold` will not be inlined, `hot` would be inlined if available.
Profile Guided Optimization (PGO): Importing Heuristics

Summary:
- `_Z3hotv`, 150 instructions, global linkage
- `_Z4coldv`, 50 instructions, global linkage

Mirror inliner heuristic by giving bonus for hot edges and a penalty for cold ones!

→ Only `cold()` would be inlined! The importer would only import `cold()`!

With PGO data, `cold` will not be inlined, `hot` would be inlined if available

→ Mirror inliner heuristic by giving bonus for hot edges and a penalty for cold ones!
ThinLTO Revisited: Incremental Build

```c
int test1(int a);
int main() {
    return test1(42);
}
```

```c
int test2(int);
int test1(int a) {
    // Do stuff
    // ...
    return test2(a+1);
}
int test1_foo(int bar) {
    return Bar-1;
}
```

**Summary:**
- `main`, 2 instructions
- `test1`, 42 instructions
- `test1_foo`, 2 instructions
- `test2`, 13 instructions

Global linkage, call `test1`

**clang -cc1-flto=thin**

```
main.o
```

```
test1.o
```

```
test2.o
```

Source file names:
- `main.cpp`
- `test1.cpp`
- `test2.cpp`

**clang -cc1-flto=thin**

```
define int @main() #0 (...
```

```
define int @Z5test1i(int) #0 (...
```

```
define int @Z9test1_fooi(int) #0 (...
```

**Summary:**
- `main.cpp`, 2 instructions
- `test1.cpp`, 42 instructions
- `test2.cpp`, 13 instructions

Global linkage, call `test1`

Global linkage, call `test1_fooi`
ThinLTO Revisited: Incremental Build

Summary:
main: 2 instructions
global linkage, call _Z5test1i

Module Hash: 4daa04c497e11d7dd51732f055

---

test1: 42 instructions
global linkage, call _Z5test1i

Module Hash: aee693ed4b5674829d05f56ef055

---

test2: 13 instructions
global linkage, call _Z5test2i

Module Hash: 4daa04c497e11d7dd51732f055
### ThinLTO Revisited: Incremental Build

**main.cpp**

```cpp
int test1(int a);
int main() {
  return test1(42);
}
```

**test1.cpp**

```cpp
int test2(int b) {
  // Do stuff
  return test2(a+1);
}
```

**test2.cpp**

```cpp
int test1_foo(int bar) {
  return Bar-1;
}
```

**Summary:**

- **main**: 2 instructions
- **_Z5test1i**: 42 instructions
- **_Z9test1_fooi**: 2 instructions
- **_Z5test2i**: 13 instructions

**Source Code:**

- **main.cpp**:
  ```cpp
define 132 @main() #0 {
    %1 = tail call 132 @Z5test1i(i32 42)
    ret 132 %1
  }
def 132 @Z5test1i(i32) #1 {
  declare 132 @Z5test1i(i32) #1
  return 132 %1
}
```

- **test1.cpp**:
  ```cpp
define 132 @Z5test2i(i32) #0 {
  %3 = add ms 132 %0, 1
  %2 = tail call 132 @Z5test2i(i32 %2)
  ret 132 %3
}
def 132 @Z9test1_fooi(i32) #2 {
  %2 = add ms 132 %0, -1
  ret 132 %2
}
```

- **test2.cpp**:
  ```cpp
define 132 @Z9test1_fooi(i32) #2 {
  %2 = add ms 132 %0, -1
  ret 132 %2
}
```

**Module Hash:**

- **main.o**: 4d9a04c497e11d7dd5184f555
- **test1.o**: aee693ed4b5674829d056ef05b
- **test2.o**: e10adb9c3508bd5e55782d884

---

**clang -cc1 -flto=thin**

---

**clang -cc1 -flto=thin**

---

**clang -cc1 -flto=thin**

---

**clang -cc1 -flto=thin**
ThinLTO Revisited: Incremental Build

**main.o**

```c
source_filename = "main.cpp"

define i32 @main() #0 {
  %1 = call i32 @Z5test1i(132)
  ret i32 %1
}

declare i32 @Z5test1i(132) #1
```

Summary:
main, 2 instructions
global linkage, call _Z5test1i

Module Hash:
4daa04c497e11d7dd51732f055

**test1.o**

```c
source_filename = "test1.cpp"

define i32 @Z5test1i(132) #0 {
  %2 = add nsw i32 %0, 1
  %3 = call i32 @Z5test2i(132 %0)
  ret i32 %3
}

define i32 @Z9test1_fooi(132) #2 {
  %0 = add nsw i32 %0, 1
  %3 = call i32 @Z9test1_fooi(132 %2)
  ret i32 %3
}

declare i32 @Z9test1_fooi(132) #1
```

Summary:
_Z5test1i, 42 instructions
global linkage, call _Z5test2i

Module Hash:
aee693ed4b5674829d05f56ef0b

**test2.o**

```c
source_filename = "test2.cpp"

define i32 @Z5test2i(132) #0 {
  ; Other stuff here...
  %3 = call i32 @Z5test1i(132 %0)
  ret i32 %3
}

define i32 @Z9test1_fooi(132) #2 {
  %0 = add nsw i32 %0, 1
  %3 = call i32 @Z9test1_fooi(132 %2)
  ret i32 %3
}

declare i32 @Z9test1_fooi(132) #1
```

Summary:
_Z5test2i, 13 instructions
global linkage, call _Z9test1_fooi

Module Hash:
e10adb9c3508bd5e55782d884
ThinLTO Revisited: Incremental Build

libLTO.dylib

main.o

```c
source_filename = "main.cpp"

void main() {(#)
    %1 = tail call 132 @Z5test1i(132)
    ret 132
}

Declare 132 @Z5test1i(132) #1

Summary:
main, 2 instructions
global linkage, call _Z5test1i

Module Hash:
4d4a04c497e11d7dd51792f055
```

test1.o

```c
source_filename = "test1.cpp"

void main() {(#)
    %2 = add now 132, %1, 1
    %3 = tail call 132 @Z9test1_fooi(132)
    ret 132
}

Define 132 @Z9test1_fooi(132) #2

Summary:
_Z5test1i, 42 instructions
global linkage, call _Z5test1i

Module Hash:
dee693ed4b5674829d0f656ef0b
```

test2.o

```c
source_filename = "test2.cpp"

void main() {(#)
    %2 = add now 132, %1, 1
    %3 = tail call 132 @Z9test1_fooi(132)
    ret 132
}

Declare 132 @Z9test1_fooi(132) #1

Summary:
_Z5test1i, 13 instructions
global linkage, call _Z59test1_fooi

Module Hash:
s10adbb9c3508b65e56792d984
```

Combined Index

- **7c897c21** → main.o, 2 instructions
  - call **3597eb0**
  - call **77c4a42**
  - call **768595e**

- **768595e** → test1.o, 2 instructions
  - call **77c4a42**

- **3597eb0** → test1.o, 42 instructions
  - call **77c4a42**

- **77c4a42** → test2.o, 13 instructions
  - call **768595e**
ThinLTO Revisited: Incremental Build

main.o

```c
source_filename = "main.cpp"

1 define i32 @main() #0 { %1 = call i32 @Z5test1i(id32) %0 ret i32 %1 }
2 %2 = add i32 %1, 1 ret i32 %2
3
4 declare i32 @Z5test1i(id32) #1
```

Summary:
main, 2 instructions
global linkage, call _Z5test1i

Module Hash:
4daa04c497e11d7dd51792f055

---

test1.o

```c
source_filename = "test1.cpp"

define i32 @Z9test1_fooi(id32) #0 {
  %2 = add i32 %0, 1
  %3 = call i32 @Z5test2i(id32 %2)
  ret i32 %3
}

1 define i32 @Z5test2i(id32) #1 {
  %2 = add i32 %0, -1
  ret i32 %2
}
2
3 declare i32 @Z5test1_fooi(id32) #1
```

Summary:
_Z5test1i, 42 instructions
global linkage, call _Z5test1i

Module Hash:
7c897c21

---

test2.o

```c
source_filename = "test2.cpp"

define i32 @Z9test1_fooi(id32) #0 {
  %2 = add i32 %0, 1
  %3 = call i32 @Z9test1_fooi(id32 %2)
  ret i32 %3
}

1 define i32 @Z9test1_fooi(id32) #1 {
  %2 = add i32 %0, -1
  ret i32 %2
}
2
3 declare i32 @Z9test1_fooi(id32) #1
```

Summary:
_Z9test2i, 13 instructions
global linkage, call _Z9test1_fooi

Module Hash:
e10adbb9c3508bd5e5656e09b

---

Combined Index

```
call 3597eb0
7c897c21  →  main.o, 2 instructions

3597eb0 → test1.o, 42 instructions
call 77c4a42
768595e  →  test1.o, 2 instructions
call 77c4a42
77c4a42  →  test2.o, 13 instructions
call 768595e
call 3597eb0
```

---

main.o

```c
source_filename = "main.cpp"

define i32 @main() #0 {
  %1 = call i32 @Z5test1i(id32) %0 ret i32 %1 
}

1 define i32 @main() #0 {
  %1 = call i32 @Z5test1i(id32) %0 ret i32 %1 
  %2 = add i32 %1, 1 ret i32 %2 
  %3 = call i32 @Z5test2i(id32 %2) ret i32 %3 
}

1 declare i32 @main() #0
```

Summary:
main, 2 instructions
global linkage, call _Z5test1i

Module Hash:
4daa04c497e11d7dd51792f055

---

test1.o

```c
source_filename = "test1.cpp"

define i32 @Z9test1_fooi(id32) #0 {
  %2 = add i32 %0, 1
  %3 = call i32 @Z5test2i(id32 %2)
  ret i32 %3 
}

1 define i32 @Z5test2i(id32) #1 {
  %2 = add i32 %0, -1
  ret i32 %2 
}
2
3 declare i32 @Z5test1_fooi(id32) #1
```

Summary:
_Z5test1i, 42 instructions
global linkage, call _Z5test1i

Module Hash:
7c897c21

---

test2.o

```c
source_filename = "test2.cpp"

define i32 @Z9test1_fooi(id32) #0 {
  %2 = add i32 %0, 1
  %3 = call i32 @Z9test1_fooi(id32 %2)
  ret i32 %3 
}

1 define i32 @Z9test1_fooi(id32) #1 {
  %2 = add i32 %0, -1
  ret i32 %2 
}
2
3 declare i32 @Z9test1_fooi(id32) #1
```

Summary:
_Z9test2i, 13 instructions
global linkage, call _Z9test1_fooi

Module Hash:
e10adbb9c3508bd5e5656e09b

---

libLTO.dylib

```c
main.o

define i32 @main() #0 {
  %1 = call i32 @Z5test1i(id32) %0 ret i32 %1 
}
```

Summary:
main, 2 instructions
global linkage, call _Z5test1i

Module Hash:
4daa04c497e11d7dd51792f055
ThinLTO Revisited: Incremental Build

libLTO.dylib

Import Lists

main.o
- 3597eb0
- 77c4a42
- 768595e

test1.o
- 3597eb0
- 77c4a42

Combined Index

- call main.o, 2 instructions
  - 7c897c21
- call test1.o, 42 instructions
  - 3597eb0
- call test2.o, 13 instructions
  - 77c4a42
  - 768595e
ThinLTO Revisited: Incremental Build

**libLTO.dylib**

**source_filename = "main.cpp"**
1. define main() {  
2. %1 = call 132 @_Z5test1i(132 %2)  
3. ret 132 %1  
4. }  
5. declare 132 @_Z5test1i(132) #1

**source_filename = "test1.cpp"**
1. define i32 @test1_fooi(i32 %2) {  
2. %3 = tail call i32 @Z5test2i(132 %2)  
3. ret i32 %3  
4. }  
5. declare i32 @test1_fooi(i32) #2

**source_filename = "test2.cpp"**
1. define i32 @test2(132) {  
2. %2 = add i32 %1  
3. %3 = tail call i32 @Z9test1_fooi(132 %2)  
4. ret %3  
5. }  
6. declare i32 @test2(132) #1

**Summary:**
- main, 2 instructions
- global linkage, call _Z5test1i

**Module Hash:**
4daa04c497e11d7dd51792f055

**Import Lists:**
- main.o
- test1.o
- test2.o
- libLTO.dylib

**Combined Index:**
- main.o → 2 instructions
  - call 3597eb0
  - 3597eb0 → test1.o → 42 instructions
  - call 77c4a42
  - 77c4a42 → test2.o → 13 instructions
  - call 768595e
  - 768595e → 2 instructions
  - test1.o → 3597eb0
  - test2.o → 768595e
  - main.o → 4daa04c497
  - test2.o → aee693ed4b
  - test1.o → e10adbb9c3
ThinLTO Revisited: Incremental Build

**libLTO.dylib**

---

**main.o**

```
source_filename = "main.cpp"

define i32 @main()% {{
  %1 = call i32 @Z5test1i(i32)
  ret i32 %1
}

declare i32 @Z5test1i(i32)
```

**test1.o**

```
source_filename = "test1.cpp"

define i32 @Z9test1_fooi()% {{
  %2 = add nsw i32 %0, 1
  %3 = call i32 @Z9test1_fooi(%2)
  %4 = ret i32 %3
}

declare i32 @Z9test1_fooi(i32)
```

**test2.o**

```
source_filename = "test2.cpp"

define i32 @Z5test2i()% {{
  %2 = add nsw i32 %0, 1
  %3 = call i32 @Z5test2i(%2)
  %4 = ret i32 %3
}

declare i32 @Z5test2i(i32)
```

---

**Summary:**
- **main**: 2 instructions
- **test1**: 42 instructions
- **test2**: 13 instructions
- **global linkage**: call _Z5test1i, call _Z9test1_fooi

**Module Hash:**
- main.o: 4daa04c497e11d7dd51792f055
- test1.o: e10adbb9c3508bd5e55782d884
- test2.o: aee693ed4b5674829d05f66e0b

---

**Import Lists**
- main.o: 3597eb0 77c4a42 768595e
e10adbb
- test1.o: e10adbb 768595e
eaee693ed
- test2.o: e10adbb

---

**Combined Index**
- main.o: 7c897c21 → main.o, 2 instructions
call 3597eb0
- test1.o: 7c897c21 → test1.o, 42 instructions
call 77c4a42 768595e
test1.o → test2.o
- test2.o: 7c897c21 → test2.o, 13 instructions
call 768595e
ThinLTO Revisited: Incremental Build

**Import Lists**

- **main.o**
  - `3597eb0`
  - `77c4a42`
  - `768595e`
- **test1.o**
  - `77c4a42`
  - `768595e`
- **test2.o**
  - `768595e`
  - `e10adbb`

**Combined Index**

- `7c897c21`: `main.o`, 2 instructions
  - call `3597eb0`
- `3597eb0`: `test1.o`, 42 instructions
  - call `77c4a42`
- `77c4a42`: `test1.o`, 2 instructions
  - call `768595e`
- `768595e`: `test1.o`, 2 instructions
  - call `77c4a42`
- `77c4a42`: `test2.o`, 13 instructions
  - call `768595e`
- `768595e`: `test1.o`, 2 instructions
  - call `e10adbb`
- `e10adbb`: `main.o`, 2 instructions
  - call `3597eb0`

**Module Hashes**

- **main.o**
  - `4daa04c497e11d7dd51792f055`
- **test1.o**
  - `aee693ed4b5674829d05f56ef0b`
- **test2.o**
  - `e10adbb9c3508bd5e55782d884`
ThinLTO Revisited: Incremental Build

**Summary:**
- **main.o**
  - 42 instructions
  - global linkage, call _Z5test1i
  - Module Hash: 4daa044d97e11d7dd51792f055

- **test1.o**
  - 2 instructions
  - global linkage, call _Z9test1_fooi
  - Module Hash: eee6936d4b5674929d05f6af0b

- **test2.o**
  - 13 instructions
  - global linkage, call _Z9test1_fooi
  - Module Hash: e10adbb9c3598be5e56792d984

**Import Lists**
- **main.o**
  - 3597eb0
  - 77c4a42
  - 768595e

- **test1.o**
  - 77c4a42
  - e10adbb

- **test2.o**
  - 768595e
  - e10adbb
ThinLTO Revisited: Incremental Build

```
main.o
1  source_filename = "main.cpp"
2
3  define i32 @main() #0 {
4    %1 = call i32 @Z5test1i(i32 %2)
5    ret i32 %1
6  }
7
8  declare i32 @Z5test1i(i32) #1

Summary:
main, 2 instructions
global linkage, call Z5test1i

Module Hash:
edaa04e497e11d7dd51792f055
```

```
test1.o
1  source_filename = "test1.cpp"
2
3  define i32 @Z9test1_fooi(i32 %2) #0 {
4    %2 = add nsw i32 %2, 1
5    %3 = call i32 @Z5test2i(i32 %2)
6    ret i32 %3
7  }
8
9  define i32 @Z5test2i(i32 %2) #2 {
10   %2 = add nsw i32 %2, -1
11   ret i32 %2
12  }

Summary:
_Z5test1i, 42 instructions
_Z9test1_fooi, 2 instructions
global linkage, call _Z5test1i
global linkage, call _Z9test1_fooi

Module Hash:
ea693ed4b5674829d05f56ef0b
```

```
test2.o
1  source_filename = "test2.cpp"
2
3  define i32 @Z9test2i(i32 %2) #0 {
4    %2 = add nsw i32 %2, 1
5    %3 = call i32 @Z9test1_fooi(i32 %2)
6    ret i32 %3
7  }
8
9  declare i32 @Z9test1_fooi(i32) #1

Summary:
_Z9test2i, 13 instructions
global linkage, call _Z9test1_fooi

Module Hash:
eaee693ed4b5674829d05f56ef0b
```

Import Lists:
- main.o: 3597eb0 77c4a42 768595e
- test1.o: e10adbb
test2.o: e10adbb

libLTO.dylib
ThinLTO O Revisited: Incremental Build
libLTO.dylib

ThinLTO Revisited: Incremental Build

Parallel ThinLTO Backends:

- Read the import list
ThinLTO Revisited: Incremental Build

Parallel ThinLTO Backends:

- Read the import list
- Compute a unique hash based on all the inputs: **without reading any IR!**
Parallel ThinLTO Backends:

- Read the import list
- Compute a unique hash based on all the inputs: **without reading any IR**!
- Query a persistent cache and early return on cache hit.
Parallel ThinLTO Backends:

- Read the import list
- Compute a unique hash based on all the inputs: **without reading any IR**!
- Query a persistent cache and early return on cache hit.
- Otherwise, process as normal, but commit the object to the cache before returning to the linker.
Parallel ThinLTO Backends:

- Read the import list
- Compute a unique hash based on all the inputs: **without reading any IR**!
- Query a persistent cache and early return on cache hit.
- Otherwise, process as normal, but commit the object to the cache before returning to the linker.
- The final native link is not incremental: all the objects are re-linked always bit-to-bit the same binary.
Phase 1: Compile

Phase 2: Thin Link

Phase 3: Backends

ThinLTO: Distributed Builds
Phase 2: Thin Link

Linker

Inter-procedural Analysis
Analyses Results

Phase 3: Backends

BE BE BE BE BE BE BE BE

ThinLTO: Distributed Builds
ThinLTO: Distributed Builds

Phase 2: Thin Link

Linker

Distributed build system

Inter-procedural Analysis

Analyses Results

Phase 3: Backends

BE BE BE BE BE BE BE BE
ThinLTO: Distributed Builds

Phase 2: Thin Link
- Inter-procedural Analysis
  - Analyses Results
  - Stream out analysis results for each backend job

Phase 3: Backends
- Opt
- BE

Distributed build system

+ Distributed build system
ThinLTO: Distributed Builds

Phase 2: Thin Link
- Linker
  - Distributed build system
  
  - Stream out analysis results for each backend job
  - Produce dependent module lists: the build system bundles these for each job

Phase 3: Backends
- Opt
- BE

Inter-procedural Analysis

Analyses Results
ThinLTO: Distributed Builds

Phase 2: Thin Link

- Thin Link
- Inter-procedural Analysis
- Analyses Results

Phase 3: Backends

- Stream out analysis results for each backend job
- Produce dependent module lists: the build system bundles these for each job
ThinLTO: Distributed Builds

Phase 2: Thin Link

Inter-procedural Analysis

Analyses Results

+ Stream out analysis results for each backend job

+ Produce dependent module lists: the build system bundles these for each job

Phase 3: Backends

Opt

BE
ThinLTO: Distributed Builds

Phase 2: Thin Link
- Inter-procedural Analysis
  - Analyses Results
  - Stream out analysis results for each backend job
  - Produce dependent module lists: the build system bundles these for each job

Phase 3: Backends
- Distributed Inter-procedural transformations based on the analyses results
- Distributed build system
- Linker

+ 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: Distributed Builds

Phase 2: Thin Link
- Distributed build system
- Linker
- Inter-procedural Analysis
- Analyses Results

Phase 3: Backends
- Opt
- BE

+ 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: Distributed Builds

Phase 2: Thin Link

- Distributed build system
- Linker

- Stream out analysis results for each backend job
- Produce dependent module lists: the build system bundles these for each job

Phase 3: Backends

- Distributed Inter-procedural transformations based on the analyses results
ThinLTO: Distributed Builds

**Phase 2: Thin Link**
- Inter-procedural Analysis
  - Analyses Results
- Linker
- Distributed build system
  - Stream out analysis results for each backend job
  - Produce dependent module lists: the build system bundles these for each job
  - Individual index files encapsulate info necessary for incremental build decisions
  - **Distributed** Inter-procedural transformations based on the analyses results

**Phase 3: Backends**
- Backends
  - Opt
  - Opt
  - Opt
  - Opt
  - Opt
  - Opt
  - Opt
- Linker
- Distributed build system
  - .bc
  - .bc
  - .bc
  - .bc

**Traditional Linking**
Performance: Link Time for Clang
Performance: Link Time for Clang

Monolithic LTO

- Release: 1,147s

Linux 16-core (32-logical)
Intel Xeon E5-2690 @ 2.90GHz
Performance: Link Time for Clang

Monolithic LTO

1 thread: 704s
2 threads: 453s
3 threads: 340s
4 threads: 275s
5 threads: 233s
6 threads: 198s
7 threads: 171s
8 threads: 153s
9 threads: 139s
10 threads: 132s
11 threads: 123s
12 threads: 117s
13 threads: 109s
14 threads: 105s
15 threads: 96s
16 threads: 96s
17 threads: 96s
18 threads: 86s
19 threads: 86s
20 threads: 93s

Release

Monolithic LTO: 1,147s
Over Monolithic LTO: x12

LLVM 3.6
LLVM 3.7
LLVM 3.8

Linux 16-core (32-logical)
Intel Xeon E5-2690 @ 2.90GHz
Performance: Link Time for Clang

LLVM 3.6
LLVM 3.7
LLVM 3.8

17x less memory!

Linux 16-core (32-logical)
Intel Xeon E5-2690 @ 2.90GHz
<table>
<thead>
<tr>
<th>Threads</th>
<th>Over Monolithic LTO:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 thread</td>
<td>x15.1</td>
</tr>
<tr>
<td>2 threads</td>
<td>x12</td>
</tr>
<tr>
<td>4 threads</td>
<td></td>
</tr>
<tr>
<td>8 threads</td>
<td></td>
</tr>
<tr>
<td>16 threads</td>
<td></td>
</tr>
<tr>
<td>41MB per thread</td>
<td>17x less memory!</td>
</tr>
</tbody>
</table>

Linux 16-core (32-logical)
Intel Xeon E5-2690 @ 2.9GHz
Performance: Link Time for Clang

Monolithic LTO

1 thread

Release: 1,147s
Line Tables: 1,199s
Total: 1,445s

2 threads

Release: 453s
Line Tables: 704s
Total: 1157s

3 threads

Release: 340s
Line Tables: 233s
Total: 573s

4 threads

Release: 275s
Line Tables: 198s
Total: 473s

5 threads

Release: 233s
Line Tables: 171s
Total: 404s

6 threads

Release: 198s
Line Tables: 153s
Total: 351s

7 threads

Release: 171s
Line Tables: 132s
Total: 291s

8 threads

Release: 153s
Line Tables: 132s
Total: 285s

9 threads

Release: 139s
Line Tables: 123s
Total: 262s

10 threads

Release: 132s
Line Tables: 123s
Total: 255s

11 threads

Release: 117s
Line Tables: 109s
Total: 226s

12 threads

Release: 109s
Line Tables: 105s
Total: 214s

13 threads

Release: 96s
Line Tables: 86s
Total: 182s

14 threads

Release: 96s
Line Tables: 86s
Total: 182s

15 threads

Release: 96s
Line Tables: 86s
Total: 182s

16 threads

Release: 96s
Line Tables: 86s
Total: 182s

17 threads

Release: 96s
Line Tables: 86s
Total: 182s

18 threads

Release: 96s
Line Tables: 86s
Total: 182s

19 threads

Release: 96s
Line Tables: 86s
Total: 182s

20 threads

Release: 93s
Line Tables: 93s
Total: 186s

41MB per thread

over Monolithic LTO: x15.1

over Monolithic LTO: x12

Linux 16-core (32-logical)
Intel Xeon E5-2690 @ 2.90GHz
Performance: Link Time for Clang

Monolithic LTO

1 thread: 1,199s
2 threads: 704s
3 threads: 453s
4 threads: 340s
5 threads: 275s
6 threads: 233s
7 threads: 198s
8 threads: 171s
9 threads: 153s
10 threads: 139s
11 threads: 132s
12 threads: 123s
13 threads: 117s
14 threads: 109s
15 threads: 105s
16 threads: 96s
17 threads: 96s
18 threads: 96s
19 threads: 86s
20 threads: 93s

Release:

1 thread: 1,147s
2 threads: 1,445s
3 threads: 1,517s
4 threads: 1,672s
5 threads: 1,872s
6 threads: 2,09s
7 threads: 2,11s
8 threads: 2,12s
9 threads: 2,23s
10 threads: 2,54s
11 threads: 2,93s
12 threads: 3,02s
13 threads: 3,02s
14 threads: 3,44s
15 threads: 3,44s
16 threads: 4,02s
17 threads: 4,65s
18 threads: 5,57s
19 threads: 6,57s
20 threads: 8,58s

Line Tables:

1 thread: 1,147s
2 threads: 1,445s
3 threads: 1,517s
4 threads: 1,672s
5 threads: 1,872s
6 threads: 2,09s
7 threads: 2,11s
8 threads: 2,12s
9 threads: 2,23s
10 threads: 2,54s
11 threads: 2,93s
12 threads: 3,02s
13 threads: 3,02s
14 threads: 3,44s
15 threads: 3,44s
16 threads: 4,02s
17 threads: 4,65s
18 threads: 5,57s
19 threads: 6,57s
20 threads: 8,58s

Monolithic LTO

- Over Monolithic LTO: x15.1
- Over Monolithic LTO: x12
- Over Monolithic LTO: ∆5.3

Linux 16-core (32-logical)
Intel Xeon E5-2690 @ 2.90GHz
Performance: Link Time for Clang

- **Monolithic LTO**
  - 1 thread: 1,147s/1,199s
  - 2 threads: 1,445s/1,517s
  - 4 threads: 2,26s
  - 5 threads: 187s
  - 6 threads: 209s
  - 7 threads: 211s
  - 8 threads: 223s
  - 9 threads: 220s
  - 10 threads: 231s
  - 11 threads: 254s
  - 12 threads: 275s
  - 13 threads: 276s
  - 14 threads: 302s
  - 15 threads: 344s
  - 16 threads: 402s
  - 17 threads: 465s
  - 18 threads: 557s
  - 19 threads: 657s
  - 20 threads: 858s

- **Over Monolithic LTO**:
  - 4 threads: x15.1
  - 8 threads: x13.7

- **Linux 16-core (32-logical)**
  - Intel Xeon E5-2690 @ 2.9GHz
  - Performance: 41MB per thread
  - Over Monolithic LTO: 5,421MB

- **Debug**
  - 273MB
  - 557MB
  - 425MB
  - 334MB
  - 190MB
  - 565MB
  - 375MB
  - 294MB
  - 42MB
  - 782MB
  - 88MB
  - 173MB
  - 1,049MB
  - 82MB
  - 883MB
  - 852MB
  - 685MB
  - 690MB
  - 540MB
  - 603MB
  - 574MB
  - 933MB
  - 572MB
  - 704s

- **Line Tables**
  - 5,372MB
  - 5,031MB
  - 4,881MB
  - 4,785MB
  - 4,471MB
  - 4,165MB
  - 3,937MB
  - 3,760MB
  - 3,609MB
  - 3,301MB
  - 2,984MB
  - 2,795MB
  - 2,494MB
  - 2,258MB
  - 2,143MB
  - 1,802MB
  - 1,682MB
  - 1,248MB
  - 933MB

- **Release**
  - 1,428MB
  - 1,403MB
  - 1,398MB
  - 1,351MB
  - 1,285MB
  - 1,240MB
  - 1,154MB
  - 1,113MB
  - 1,049MB
  - 968MB
  - 952MB
  - 895MB
  - 852MB
  - 685MB
  - 690MB
  - 603MB
  - 497MB
  - 391MB
  - 302MB

- **Divided by**:
  - 5.3
  - 7.5

**Debug Size**

- 20 threads: 9,678MB
- 19 threads: 933MB
- 18 threads: 86s
- 17 threads: 86s
- 16 threads: 86s
- 15 threads: 93s
- 14 threads: 96s
- 13 threads: 96s
- 12 threads: 96s
- 11 threads: 96s
- 10 threads: 96s
- 9 threads: 96s
- 8 threads: 96s
- 7 threads: 96s
- 6 threads: 96s
- 5 threads: 96s
- 4 threads: 96s
- 3 threads: 96s
- 2 threads: 96s
- 1 thread: 96s

**54MB per thread**
Performance: Link Time for Clang

Linux 16-core (32-logical)
Intel Xeon E5-2690 @ 2.9GHz

Monolithic LTO
1 thread
2 threads
3 threads
4 threads
5 threads
6 threads
7 threads
8 threads
9 threads
10 threads
11 threads
12 threads
13 threads
14 threads
15 threads
16 threads
17 threads
18 threads
19 threads
20 threads

Monolithic LTO: x15.1 x12 x10.8

41MB per thread
54MB per thread

Release
Line Tables
Debug

Performance: Link Time for Clang

LLVM 3.6
LLVM 3.7
LLVM 3.8

41MB per thread
54MB per thread

÷5.3 ÷7.5

Release
Line Tables
Debug

5,421MB
9,678MB

17,351MB
Performance: Link Time for Clang

- **Monolithic LTO**
  - 1 thread: 1.147s, 1.190s, 1.500s, 1.545s, 1.117s, 2.643s
  - 2 threads: 274s, 343s, 253s, 293s, 233s, 243s
  - 3 threads: 171s, 166s, 156s
  - 4 threads: 139s, 149s, 149s, 141s
  - 5 threads: 123s, 132s
  - 6 threads: 117s, 120s
  - 7 threads: 109s, 119s
  - 8 threads: 106s
  - 9 threads: 104s
  - 10 threads: 104s
  - 11 threads: 103s
  - 12 threads: 102s
  - 13 threads: 102s
  - 14 threads: 102s
  - 15 threads: 101s
  - 16 threads: 99s
  - 17 threads: 98s
  - 18 threads: 98s
  - 19 threads: 96s
  - 20 threads: 95s

- **Release**
  - 1 thread: 17.351MB
  - 2 threads: 1,428MB
  - 4 threads: 1,144MB
  - 8 threads: 933MB
  - 16 threads: 574MB

- **Line Tables**
  - 1 thread: 5,372MB
  - 2 threads: 4,344MB
  - 4 threads: 3,301MB
  - 8 threads: 2,334MB
  - 16 threads: 1,315MB

- **Debug**
  - 1 thread: 933MB
  - 2 threads: 1,115MB
  - 4 threads: 1,035MB

**30x less memory!**

- **41MB per thread**
- **54MB per thread**

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

- **Performance: Link Time for Clang**
  - Over Monolithic LTO: x15.1, x13.7
  - Over Monolithic LTO: ÷5.3, ÷7.5
Performance: Link Time for Clang

Monolithic LTO

<table>
<thead>
<tr>
<th>Threads</th>
<th>Release (s)</th>
<th>Debug (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.147</td>
<td>1.190</td>
</tr>
<tr>
<td>2</td>
<td>1.500</td>
<td>1.517</td>
</tr>
<tr>
<td>3</td>
<td>2.643</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3.144</td>
<td>3.468</td>
</tr>
<tr>
<td>5</td>
<td>3.858</td>
<td>4.152</td>
</tr>
<tr>
<td>6</td>
<td>4.657</td>
<td>4.951</td>
</tr>
<tr>
<td>7</td>
<td>5.576</td>
<td>5.869</td>
</tr>
<tr>
<td>8</td>
<td>6.575</td>
<td>6.958</td>
</tr>
<tr>
<td>9</td>
<td>7.655</td>
<td>8.058</td>
</tr>
<tr>
<td>10</td>
<td>8.857</td>
<td>9.258</td>
</tr>
<tr>
<td>11</td>
<td>10.156</td>
<td>10.558</td>
</tr>
<tr>
<td>12</td>
<td>11.555</td>
<td>11.958</td>
</tr>
<tr>
<td>13</td>
<td>12.957</td>
<td>13.358</td>
</tr>
<tr>
<td>14</td>
<td>14.456</td>
<td>14.858</td>
</tr>
<tr>
<td>15</td>
<td>16.056</td>
<td>16.458</td>
</tr>
<tr>
<td>16</td>
<td>17.655</td>
<td>18.058</td>
</tr>
<tr>
<td>17</td>
<td>19.256</td>
<td>19.658</td>
</tr>
<tr>
<td>18</td>
<td>20.856</td>
<td>21.258</td>
</tr>
<tr>
<td>19</td>
<td>22.457</td>
<td>22.858</td>
</tr>
<tr>
<td>20</td>
<td>24.056</td>
<td>24.458</td>
</tr>
</tbody>
</table>

Over Monolithic LTO:
- x15.1
- x13.7
- x12.5

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

41MB per thread
54MB per thread
228MB per thread
Performance: Link Time for Clang

<table>
<thead>
<tr>
<th>Threads</th>
<th>Release</th>
<th>Line Tables</th>
<th>Debug</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 thread</td>
<td>1,147s</td>
<td>1,190s</td>
<td>1,500s</td>
</tr>
<tr>
<td>2 threads</td>
<td>704s</td>
<td></td>
<td>1,315s</td>
</tr>
<tr>
<td>3 threads</td>
<td></td>
<td></td>
<td>2,643s</td>
</tr>
<tr>
<td>4 threads</td>
<td>340s</td>
<td>469s</td>
<td>858s</td>
</tr>
<tr>
<td>5 threads</td>
<td>275s</td>
<td>293s</td>
<td>557s</td>
</tr>
<tr>
<td>6 threads</td>
<td>233s</td>
<td>249s</td>
<td>465s</td>
</tr>
<tr>
<td>7 threads</td>
<td>190s</td>
<td>212s</td>
<td>402s</td>
</tr>
<tr>
<td>8 threads</td>
<td>153s</td>
<td>169s</td>
<td>344s</td>
</tr>
<tr>
<td>9 threads</td>
<td>139s</td>
<td>149s</td>
<td>302s</td>
</tr>
<tr>
<td>10 threads</td>
<td>117s</td>
<td>135s</td>
<td>276s</td>
</tr>
<tr>
<td>11 threads</td>
<td>103s</td>
<td>115s</td>
<td>269s</td>
</tr>
<tr>
<td>12 threads</td>
<td>123s</td>
<td>132s</td>
<td>254s</td>
</tr>
<tr>
<td>13 threads</td>
<td>119s</td>
<td>129s</td>
<td>231s</td>
</tr>
<tr>
<td>14 threads</td>
<td>109s</td>
<td>119s</td>
<td>220s</td>
</tr>
<tr>
<td>15 threads</td>
<td>106s</td>
<td>116s</td>
<td>215s</td>
</tr>
<tr>
<td>16 threads</td>
<td>96s</td>
<td>111s</td>
<td>212s</td>
</tr>
<tr>
<td>17 threads</td>
<td>88s</td>
<td>102s</td>
<td>211s</td>
</tr>
<tr>
<td>18 threads</td>
<td>90s</td>
<td>103s</td>
<td>209s</td>
</tr>
<tr>
<td>19 threads</td>
<td>86s</td>
<td>98s</td>
<td>209s</td>
</tr>
<tr>
<td>20 threads</td>
<td>81s</td>
<td>92s</td>
<td>226s</td>
</tr>
</tbody>
</table>

Over Monolithic LTO:
- x15.1
- x13.7
- x12.5

Linux 16-core (32-logical)
Intel Xeon E5-2690 @ 2.90GHz
Comparison with GCC LTO
Comparison with GCC LTO

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

2. LTRANS: Parallel backends performing inlining within each partition, plus usual optimizations and code generation
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
Scaling with the Input Size

- LLVM/Clang - C/C++ Compiler
- Chromium - open-source web browser
- Ad Delivery - internal Google datacenter application
Scaling with the Input Size

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

Number of IR Files

- Clang: 1,938
- Chromium: 17,798
- Ad Delivery: 13,829
Scaling with the Input Size

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

Number of IR Files

<table>
<thead>
<tr>
<th></th>
<th>Clang</th>
<th>Chromium</th>
<th>Ad Delivery</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>1,938</td>
<td>17,798</td>
<td>13,829</td>
</tr>
</tbody>
</table>

Total IR Size (MB)

<table>
<thead>
<tr>
<th></th>
<th>Clang</th>
<th>Chromium</th>
<th>Ad Delivery</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size in MB</td>
<td>217</td>
<td>706</td>
<td>1,073</td>
</tr>
</tbody>
</table>
Scaling with the Input Size

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

<table>
<thead>
<tr>
<th></th>
<th>Number of IR Files</th>
<th>Total IR Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Clang</td>
<td>Chromium</td>
</tr>
<tr>
<td>Clang</td>
<td>1,938</td>
<td>17,798</td>
</tr>
<tr>
<td>Chromium</td>
<td>17,798</td>
<td></td>
</tr>
<tr>
<td>Ad Delivery</td>
<td>13,829</td>
<td>706</td>
</tr>
</tbody>
</table>
Scaling with the Input Size

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

Number of IR Files:
- Clang: 1,938
- Chromium: 17,798
- Ad Delivery: 13,829

Total IR Size (MB):
- Clang: 3,554
- Chromium: 7,544
- Ad Delivery: 7,469

Call/Reference Graph Size:
- Clang: 217
- Chromium: 706
- Ad Delivery: 1,073

Nodes: 169k
- Clang: 169k
- Chromium: 858k
- Ad Delivery: 1,953k
Scaling with the Input Size

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

### Number of IR Files
- Clang: 1,938
- Chromium: 17,798
- Ad Delivery: 13,829

### Total IR Size (MB)

<table>
<thead>
<tr>
<th></th>
<th>-g0</th>
<th>-g2</th>
<th>Total IR Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clang</td>
<td>217</td>
<td>706</td>
<td>7,544</td>
</tr>
<tr>
<td>Chromium</td>
<td>3,554</td>
<td></td>
<td>7,469</td>
</tr>
<tr>
<td>Ad Delivery</td>
<td>1,073</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Call/Reference Graph Size

<table>
<thead>
<tr>
<th></th>
<th>Nodes</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clang</td>
<td>169k</td>
<td>414k</td>
</tr>
<tr>
<td>Chromium</td>
<td>858k</td>
<td>1,953k</td>
</tr>
<tr>
<td>Ad Delivery</td>
<td>2,403k</td>
<td>6,809k</td>
</tr>
</tbody>
</table>
Serial Step Comparisons

*Thin Link vs GCC WPA*

Linux 16-core (32-logical) Intel Xeon E5-2690 @ 2.90GHz
Serial Step Comparisons
Thin Link vs GCC WPA

<table>
<thead>
<tr>
<th>Serial Step</th>
<th>Clang</th>
<th>Chromium</th>
<th>Ad Delivery</th>
</tr>
</thead>
<tbody>
<tr>
<td>ThinLTO</td>
<td>0.13</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>GCC WPA -g0</td>
<td>2.27</td>
<td>1.99</td>
<td>8.05</td>
</tr>
<tr>
<td>GCC WPA -g2</td>
<td>2.89</td>
<td>10.77</td>
<td>26</td>
</tr>
</tbody>
</table>

Peak Memory (GB)

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

Thin Link vs GCC WPA

<table>
<thead>
<tr>
<th>Clang</th>
<th>Chromium</th>
<th>Ad Delivery</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.13</td>
<td>1</td>
<td>180m 0s</td>
</tr>
<tr>
<td>2.27</td>
<td>1.99</td>
<td>1m 42s</td>
</tr>
<tr>
<td>8.05</td>
<td>2.89</td>
<td>5m 0s</td>
</tr>
<tr>
<td>25</td>
<td>10.77</td>
<td>4m 0s</td>
</tr>
<tr>
<td>26</td>
<td>8.05</td>
<td>0m 30s</td>
</tr>
<tr>
<td>1</td>
<td>2.89</td>
<td>1m 16s</td>
</tr>
<tr>
<td>25</td>
<td>10.77</td>
<td>0m 55s</td>
</tr>
<tr>
<td>26</td>
<td>8.05</td>
<td>0m 5s</td>
</tr>
</tbody>
</table>

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

No Debug (-g0)

Serial Phase  Parallel Phase

Table:

<table>
<thead>
<tr>
<th>Monolithic LTO</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>8</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>ThinLTO</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1/16</td>
<td>1/16</td>
<td>1/32</td>
</tr>
</tbody>
</table>

GCC: WPA+LTRANS

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

No Debug (-g0)

Monolithic LTO

28m18s

Serial Phase

Parallel Phase

Time

GCC: WPA+LTRANS
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
Chromium Build Comparisons

No Debug (-g0)

Time

Monolithic LTO 8 ThinLTO 16 32

28m18s 6m 3m38s 2m48

Serial Phase Parallel Phase

GCC: WPA+LTRANS

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

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
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
Distributed Build: Ad Delivery

- ThinLTO
- O2

Graph showing the relationship between the number of nodes and time (in seconds). The graph indicates that both ThinLTO and O2 improve with an increase in the number of nodes, with ThinLTO generally showing a slight improvement over O2.
Performance: **Incremental** Build Time for Clang

*Time to run `ninja clang`*
Performance: **Incremental** Build Time for Clang

Time to run `ninja clang`

- Monolithic LTO: 970s
- Thin LTO: 315s
- Non LTO: 252s

Clean Build

2013 MacPro - 12-cores E5-2697 @ 2.70GHz
Performance: **Incremental** Build Time for Clang

*Time to run `ninja clang`*

- Monolithic LTO: 970s
- Thin LTO: 315s (+25%)
- Non LTO: 252s

*2013 MacPro - 12-cores E5-2697 @ 2.70GHz*
Performance: Incremental Build Time for Clang

Time to run `ninja clang`

- **Monolithic LTO**
- **Thin LTO**
- **Non LTO**

Clean Build: 970s

- Incremental Build Time for Clang

Changing the implementation of DenseMap::grow():

- +25% change

2013 MacPro - 12-cores E5-2697 @ 2.70GHz
Performance: **Incremental** Build Time for Clang

Time to run `ninja clang`

- **Monolithic LTO**: 970s
- **Thin LTO**: 960s
- **Non LTO**:
  - Clean Build: 315s (+25%)
  - Changing the implementation of DenseMap::grow(): 302s
  - 247s

2013 MacPro - 12-cores E5-2697 @ 2.70GHz
Performance: **Incremental** Build Time for Clang

**Time to run `ninja clang`**

- **Monolithic LTO**: 970s
- **Thin LTO**: 960s
- **Non LTO**: 734s

---

<table>
<thead>
<tr>
<th>Clean Build</th>
<th>Monolithic LTO</th>
<th>Thin LTO</th>
<th>Non LTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>970s</td>
<td>315s (+25%)</td>
<td>252s</td>
<td>247s</td>
</tr>
</tbody>
</table>

- **Changing the implementation of DenseMap::grow():** 302s vs. 252s (+25%)
- **Changing the implementation of visitCallInst() in InstCombineCalls.cpp:** 247s vs. 252s (-2%)

2013 MacPro - 12-cores E5-2697 @ 2.70GHz
Performance: **Incremental** Build Time for Clang

Time to run `ninja clang`

- **Monolithic LTO**: 970s
- **Thin LTO**: 960s
- **Non LTO**: 734s

### Clean Build
- **Monolithic LTO**: 315s
- **Thin LTO**: 252s (25% faster than Monolithic)
- **Non LTO**: 970s

### Changing the implementation of DenseMap::grow()
- **Monolithic LTO**: 302s
- **Thin LTO**: 247s
- **Non LTO**: 960s

### Changing the implementation of visitCallInst() in InstCombineCalls.cpp
- **Monolithic LTO**: 302s
- **Thin LTO**: 247s
- **Non LTO**: 960s

2013 MacPro - 12-cores E5-2697 @ 2.70GHz
Performance: **Incremental** Build Time for Clang

**Time to run `ninja clang`**

- **Monolithic LTO**: 970s
- **Thin LTO**: 960s
- **Non LTO**: 734s

**Clean Build**
- 315s (252s = +25%)

**Changing the implementation of DenseMap::grow()**
- 302s
- 247s

**Changing the implementation of visitCallInst() in InstCombineCalls.cpp**
- 960s

2013 MacPro - 12-cores E5-2697 @ 2.70GHz
Performance: **Incremental** Build Time for Clang

Time to run `ninja clang`

- **Monolithic LTO**
  - Clean Build: 970s
  - Changing the implementation of `DenseMap::grow()`: 315s (+25%)
  - Changing the implementation of `visitCallInst()` in `InstCombineCalls.cpp`: 302s

- **Thin LTO**
  - Clean Build: 960s
  - Changing the implementation of `DenseMap::grow()`: 247s

- **Non LTO**
  - Changing the implementation of `visitCallInst()` in `InstCombineCalls.cpp`: 734s

---

2013 MacPro - 12-cores E5-2697 @ 2.70GHz
Performance against Monolithic LTO: LLVM test-suite
Performance against Monolithic LTO: LLVM test-suite

Runtime on X86_64
(similar results observed on ARM64)
Performance against Monolithic LTO: LLVM test-suite

Runtime on X86_64
(similar results observed on ARM64)

Some benchmarks are stressing global variables
Performance against Monolithic LTO: LLVM test-suite

Runtime on X86_64
(similar results observed on ARM64)

Some benchmarks are stressing global variables

Benefits from a more aggressive LTO pass pipeline
Performance against Monolithic LTO: LLVM test-suite

Runtime on X86_64
(similar results observed on ARM64)

-60%
-45%
-30%
-15%
0%
15%
30%
45%
60%

Some benchmarks are stressing global variables

Benefits from a more aggressive LTO pass pipeline

Binary Sizes on ARM64 -Os

-10%
0%
10%
20%
30%
40%

~3.25% regression GeoMean (but still ~2.6% improvement over non-LTO)

Mostly due to lack of hidden visibility, can be recovered by improving implementation
Run-time Performance: SPEC cpu2006
Improvement over -O2 (all with PGO)

- Monolithic LTO + PGO
- ThinLTO + PGO
Future Work
Future Work

• On-going new libLTO Interface: better linker information for better codegen
  Example: resolution of weak definition to strong definition.
Future Work

• On-going new libLTO Interface: better linker information for better codegen
  Example: resolution of weak definition to strong definition.
• On-going tuning for PGO
Future Work

- On-going new libLTO Interface: better linker information for better codegen
  Example: resolution of weak definition to strong definition.
- On-going tuning for PGO
- LTO Devirtualization and CFI (Control-Flow Integrity) integration.
Future Work

• On-going new libLTO Interface: better linker information for better codegen
  Example: resolution of weak definition to strong definition.
• On-going tuning for PGO
• LTO Devirtualization and CFI (Control-Flow Integrity) integration.
• Propagate function attributes across modules via summary.
Future Work

- On-going new libLTO Interface: better linker information for better codegen
  Example: resolution of weak definition to strong definition.
- On-going tuning for PGO
- LTO Devirtualization and CFI (Control-Flow Integrity) integration.
- Propagate function attributes across modules via summary.
- More bitcode format changes to improve link-time, especially with debug info
Future Work

- On-going new libLTO Interface: better linker information for better codegen. Example: resolution of weak definition to strong definition.
- On-going tuning for PGO
- LTO Devirtualization and CFI (Control-Flow Integrity) integration.
- Propagate function attributes across modules via summary.
- More bitcode format changes to improve link-time, especially with debug info.
- Ability to move function across modules instead of importing, especially useful when a single call site exists and we could internalize.
Future Work

- On-going new libLTO Interface: better linker information for better codegen. Example: resolution of weak definition to strong definition.
- On-going tuning for PGO
- LTO Devirtualization and CFI (Control-Flow Integrity) integration.
- Propagate function attributes across modules via summary.
- More bitcode format changes to improve link-time, especially with debug info.
- Ability to move function across modules instead of importing, especially useful when a single call site exists and we could internalize.
- *Augment* the edges in the call-graph with mod-ref, constant range, …
Special Thanks:
Akira Hatanaka and Bruno Cardoso Lopes (*performance testing/debunking*),
Adrian Prantl and Duncan Exon Smith (*major changes in debug info and bitcode format*),
Peter Collingbourne (new LTO API)
Rafael Ávila de Espíndola (Lot of support and help!)
  Davide Italiano (lld support)
  Piotr Padlewski (PGO import heuristics)
Conclusion

Special Thanks:
Akira Hatanaka and Bruno Cardoso Lopes (*performance testing/debunking*),
Adrian Prantl and Duncan Exon Smith (*major changes in debug info and bitcode format*),
Peter Collingbourne (new LTO API)
Rafael Ávila de Espíndola (Lot of support and help!)
Davide Italiano (lld support)
Piotr Padlewski (PGO import heuristics)
Conclusion

Scales as a regular non-LTO build

Special Thanks:
Akira Hatanaka and Bruno Cardoso Lopes (*performance testing/debunking*),
Adrian Prantl and Duncan Exon Smith (*major changes in debug info and bitcode format*),
Peter Collingbourne (new LTO API)
Rafael Ávila de Espíndola (Lot of support and help!)
Davide Italiano (lld support)
Piotr Padlewski (PGO import heuristics)
Conclusion

Scales as a regular non-LTO build

Performance close to Monolithic LTO

Special Thanks:

Akira Hatanaka and Bruno Cardoso Lopes (*performance testing/debunking*),
Adrian Prantl and Duncan Exon Smith (*major changes in debug info and bitcode format*),
Peter Collingbourne (new LTO API)
Rafael Ávila de Espíndola (Lot of support and help!)
Davide Italiano (lld support)
Piotr Padlewski (PGO import heuristics)
Conclusion

Scales as a regular non-LTO build

Performance close to Monolithic LTO

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

Special Thanks:
Akira Hatanaka and Bruno Cardoso Lopes *(performance testing/debunking)*,
Adrian Prantl and Duncan Exon Smith *(major changes in debug info and bitcode format)*,
Peter Collingbourne *(new LTO API)*
Rafael Ávila de Espíndola *(Lot of support and help!)*
Davide Italiano *(lld support)*
Piotr Padlewski *(PGO import heuristics)*
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

Special Thanks:
Akira Hatanaka and Bruno Cardoso Lopes (performance testing/debunking),
Adrian Prantl and Duncan Exon Smith (major changes in debug info and bitcode format),
Peter Collingbourne (new LTO API)
Rafael Ávila de Espíndola (Lot of support and help!)
Davide Italiano (lld support)
Piotr Padlewski (PGO import heuristics)
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

Special Thanks:
Akira Hatanaka and Bruno Cardoso Lopes (performance testing/debunking),
Adrian Prantl and Duncan Exon Smith (major changes in debug info and bitcode format),
Peter Collingbourne (new LTO API)
Rafael Ávila de Espíndola (Lot of support and help!)
Davide Italiano (Ild support)
Piotr Padlewski (PGO import heuristics)
Backup Slides
Different optimization pipelines

-O3 LTO
- O3 w/o LTO
- O3 ThinLTO

CGSCC

{Drop Available Ext.}

Inliner

IR Simplification

Early Opt

Aggressive Optimizations

Codegen
Different optimization pipelines

- **-O3 LTO**
  - Early Opt
  - IR Simplification
  - Inliner
  - Aggressive Optimizations

- **-O3 w/o LTO**
  - Early Opt
  - IR Simplification
  - Inliner
  - Drop Available Ext.
  - Aggressive Optimizations

- **-O3 ThinLTO**
  - Early Opt
  - IR Simplification
  - Inliner

**CodeGen**
Different optimization pipelines

- **-O3 LTO**
  - Early Opt
  - IR Simplification
  - Inliner
  - Aggressive Optimizations

- **-O3 w/o LTO**
  - Early Opt
  - IR Simplification
  - Inliner
  - Drop Available Ext.
  - Aggressive Optimizations

- **-O3 ThinLTO**
  - Early Opt
  - IR Simplification
  - Inliner
  - Aggressive Optimizations

Linker

CodeGen
Different optimization pipelines

- **-O3 LTO**
  - Early Opt
  - IR Simplification
  - Inliner
  - Aggressive Optimizations

- **-O3 w/o LTO**
  - Early Opt
  - IR Simplification
  - Inliner
  - Drop Available Ext.
  - Aggressive Optimizations

- **-O3 ThinLTO**
  - Early Opt
  - IR Simplification
  - Inliner
  - Custom LTO PIPELINE (includes inliner, ...)

Linker

CodeGen
Different optimization pipelines

- **-O3 LTO**
  - Early Opt
  - IR Simplification
  - Inliner
  - Aggressive Optimizations
  - Custom LTO PIPELINE (includes inliner, ...)

- **-O3 w/o LTO**
  - Early Opt
  - IR Simplification
  - Inliner
  - Aggressive Optimizations

- **-O3 ThinLTO**
  - Early Opt
  - IR Simplification
  - Inliner
  - Drop Available Ext.
  - Aggressive Optimizations
Different optimization pipelines

**-O3 LTO**
- Early Opt
- IR Simplification
- Inliner
- Aggressive Optimizations
- Custom LTO PIPELINE (includes inliner, ...)

**-O3 w/o LTO**
- Early Opt
- IR Simplification
- Inliner
- Drop Available Ext.
- Aggressive Optimizations

**-O3 ThinLTO**
- Early Opt
- IR Simplification
- Inliner

Linker

CodeGen
Different optimization pipelines

- **-O3 LTO**
  - Early Opt
  - IR Simplification
  - Inliner
  - Aggressive Optimizations

- **-O3 w/o LTO**
  - Early Opt
  - IR Simplification
  - Inliner
  - Drop Available Ext.
  - Aggressive Optimizations

- **-O3 ThinLTO**
  - Early Opt
  - IR Simplification
  - Inliner
  - Linker

Custom LTO PIPELINE (includes inliner, ...)

CodeGen

CodeGen

CodeGen
Different optimization pipelines

**-O3 LTO**
- Early Opt
- IR Simplification
- Inliner
- Aggressive Optimizations

**Custom LTO PIPELINE** (includes inliner, ...)

---

**-O3 w/o LTO**
- Early Opt
- IR Simplification
- Inliner
- Drop Available Ext.
- Aggressive Optimizations

---

**-O3 ThinLTO**
- Early Opt
- IR Simplification
- Inliner
- Drop Available Ext.
- Aggressive Optimizations
Different optimization pipelines

- **-O3 LTO**
  - Early Opt
  - IR Simplification
  - Inliner
  - Aggressive Optimizations
  - Custom LTO PIPELINE (includes inliner, …)
  - Linker
  - Codegen

- **-O3 w/o LTO**
  - Early Opt
  - IR Simplification
  - Inliner
  - Drop Available Ext.
  - Aggressive Optimizations
  - Linker
  - Codegen

- **-O3 ThinLTO**
  - Early Opt
  - IR Simplification
  - Inliner
  - Drop Available Ext.
  - Aggressive Optimizations
  - Linker
  - Codegen
clang:

input files 217MB, and final binary is 52M

Summary blocks 0.85% on avg of IR .o sizes

Combined index total: 4M on-disk
(not stored on disk unless distributed though)
Chromium Build Comparisons

*With Debug (-g2)*
Chromium Build Comparisons

With Debug (-g2)
Chromium Build Comparisons

With Debug (-g2)

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

16m 30s
22m 0s
16m 30s
11m 0s
5m 30s
0m 0s

> 25m

LLVM LTO
ThinLTO
GCC: WPA+LTRANS

Serial portion
Parallel portion

Crash
Chromium Build Comparisons
With Debug (-g2)

-> For GCC, time versus memory tradeoff is much more severe!
Run-time Performance: SPEC cpu2006

Improvement over -O2 (no PGO)
<table>
<thead>
<tr>
<th>Type</th>
<th>LTO</th>
<th>ThinLTO</th>
<th>Increase</th>
<th>LTO</th>
<th>ThinLTO</th>
<th>Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>-g0</td>
<td>4276168</td>
<td>4438720</td>
<td>3.80%</td>
<td>4087383</td>
<td>4161320</td>
<td>1.81%</td>
</tr>
<tr>
<td>-g</td>
<td>12347112</td>
<td>20794136</td>
<td>68.41%</td>
<td>4087655</td>
<td>4161456</td>
<td>1.81%</td>
</tr>
</tbody>
</table>
### Binary/Text Size: SPEC cpu2006 471.gcc

<table>
<thead>
<tr>
<th>Type</th>
<th>Binary (MB)</th>
<th>Text (MB)</th>
<th>Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LTO</td>
<td>ThinLTO</td>
<td>Increase</td>
</tr>
<tr>
<td>-g0</td>
<td>4276168</td>
<td>4438720</td>
<td>3.80%</td>
</tr>
<tr>
<td>-g</td>
<td>12347112</td>
<td>20794136</td>
<td>68.41%</td>
</tr>
</tbody>
</table>

- Despite recent improvements, still too much debug metadata imported during ThinLTO importing!
- E.g. composite type definitions
- Area of future work