What happens after compiling?

葛天傲
getao3@mail2.sysu.edu.cn

DCS290, 04/22/2021

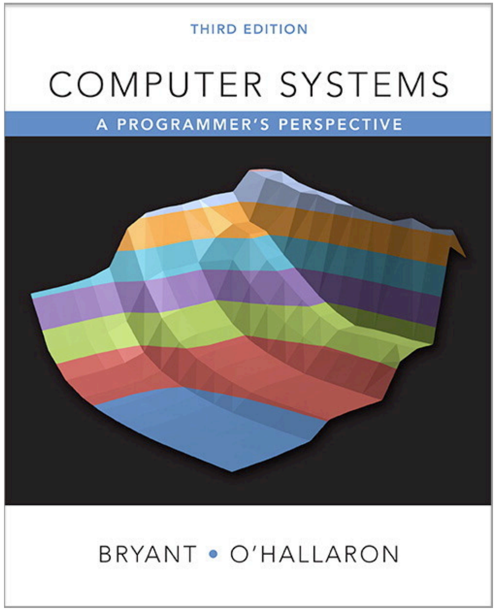
中山大學 SUN YAT-SEN UNIVERSITY
NSCC GZ 国家超级计算广州中心 NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

What happens after compiling?

References

- Computer Systems: A Programmer’s Perspective (CSAPP), Bryant and O’Hallaron（深入理解计算机系统）

See https://www.cs.cmu.edu/~213/ for more



THIRD EDITION

COMPUTER SYSTEMS

A PROGRAMMER’S PERSPECTIVE

BRYANT • O’HALLARON

Phases of a compiler

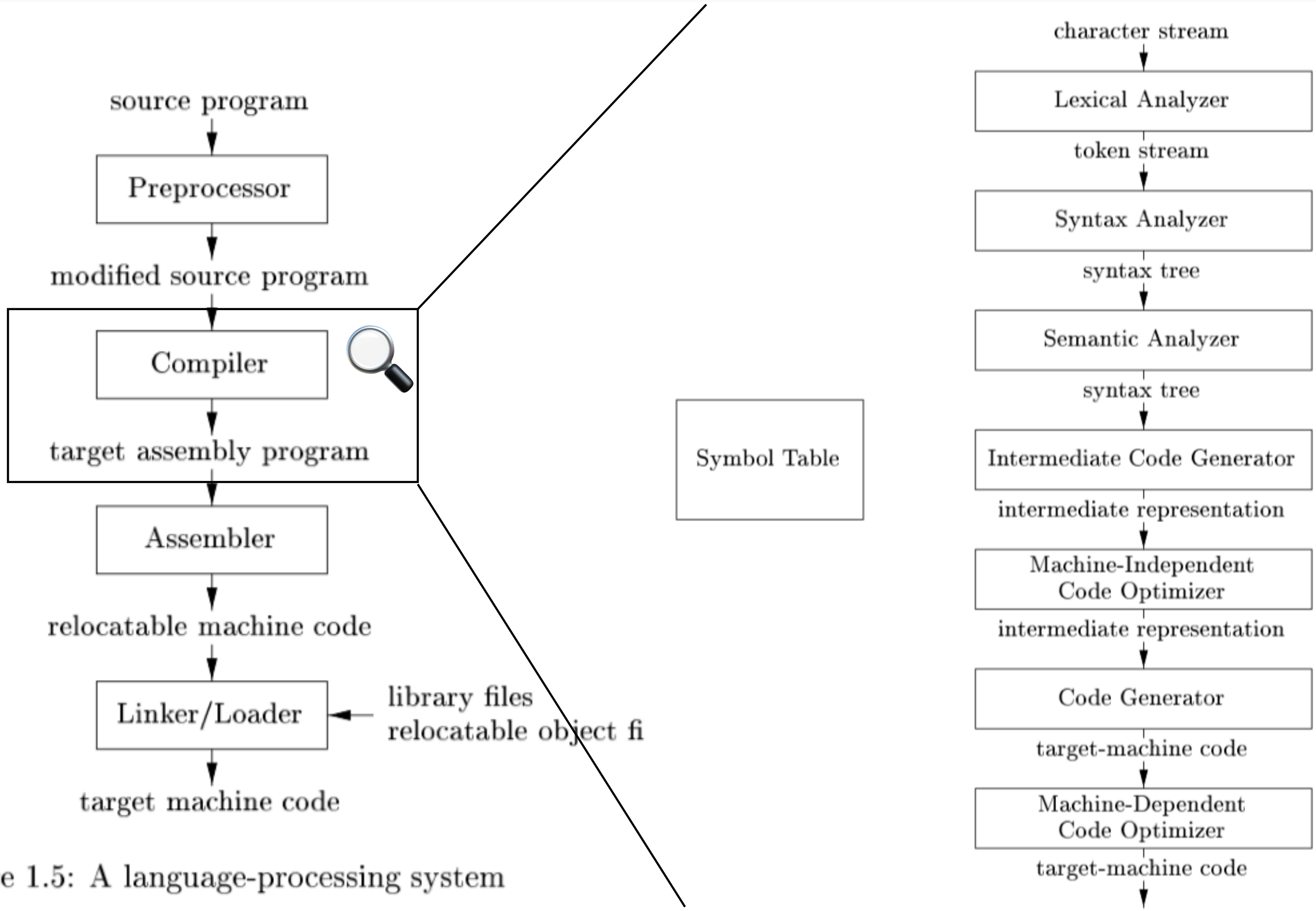


Figure 1.5: A language-processing system

Figure 1.6: Phases of a compiler

What we will talk about today

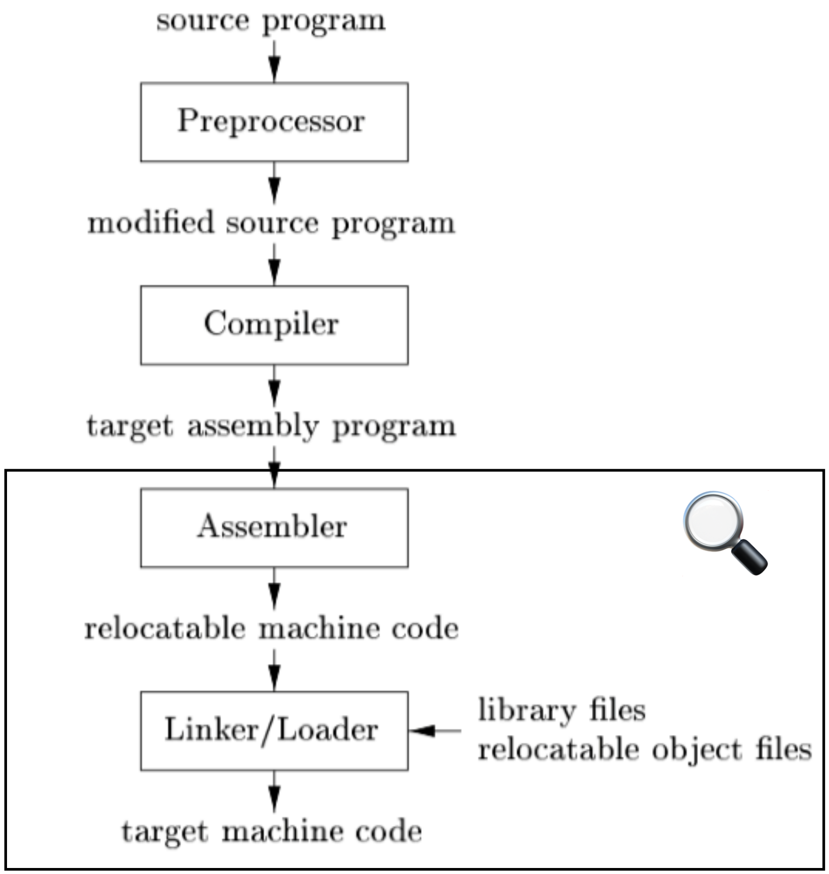


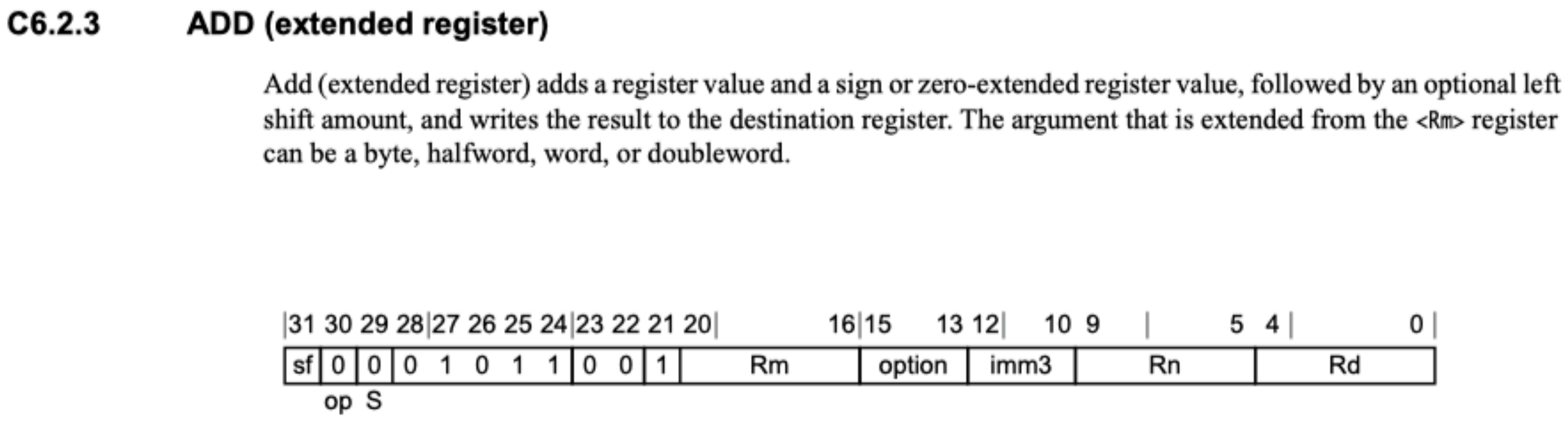
Figure 1.5: A language-processing system

Focus on Assembler and Linker.

- Why we need an assembler?
- Is an assembler necessary?
- Can we do optimizations when assembling?
- Why we need a linker?
- What do Linkers do?
- Can we do optimizations when linking?
- How does a program run and can optimize more?

Why we need an assembler（汇编器）？

Question: If compiler converts high-level language to machine code, why do we even need assembler?



C6.2.3 ADD (extended register)

Add (extended register) adds a register value and a sign or zero-extended register value, followed by an optional left shift amount, and writes the result to the destination register. The argument that is extended from the <Rm> register can be a byte, halfword, word, or doubleword.

31	30	29	28	27	26	25	24	23	22	21	20 16	15 13	12 10	9 5	4 0
sf	0	0	0	1	0	1	1	0	0	1	Rm	option	imm3	Rn	Rd
	op	S													

-- Arm Architecture Reference Manual®

- An Easy way for human to talk about / analyze machine code.
- Write/debug compilers for a CPU architecture more easily.

How was the first compiler compiled? --Stack Overflow

Is an assembler necessary?

Necessary, but we can **integrate** assembler into compiler.

Outside of GCC, the mainstream x86 C and C++ compilers (clang/LLVM, MSVC, ICC) go straight to machine code, with the option of printing asm text if you ask them to.

The integrated assembler when used with the rest of the LLVM system allows source to be compiled directly to a native object file without the need of outputting assembly instructions to a file and then parsing them back in order to encode them.

This provides the benefit of faster compiling, and when combined with the C language compiler clang, allows C/C++ to native object file compilation in one step ready for linking.

So why GCC still emits assembly code?

Depend on the design of compiler.

Can we do optimizations when assembling?

Sure, but sometimes **trivial.**

The primary point of assembly language is that what you write translates directly to individual machine instructions. Nearly *all* optimization is up to you, the programmer.

Redundant Zero Extension:

```
andl $255,%eax
mov %eax,%eax
```

Redundant Memory Access：

```
movq 24(%rsp), %rdx
movq 24(%rsp), %rcx
```

from MAO -- An extensible micro-architectural optimizer. CGO '11

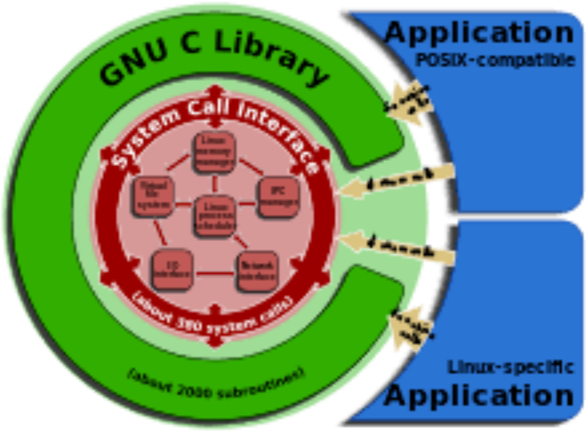
Why we need a linker（链接器）？

Reason 1: Modularity（模块化）

- Program can be written as a collection of smaller source files, rather than one monolithic mass.

- Can build libraries of common functions (more on this later)

  e.g., Math library, standard C library



Why we need a linker? (cont)

Reason 2: Efficiency

- Time: Separate compilation
  - Change one source file, compile, and then relink.
  - No need to recompile other source files.
    e.g., Make, CMake
- Space: Libraries
  - Common functions can be aggregated into a single file...
  - Yet executable files and running memory images contain only code for the functions they actually use.

GNU Make

CMake

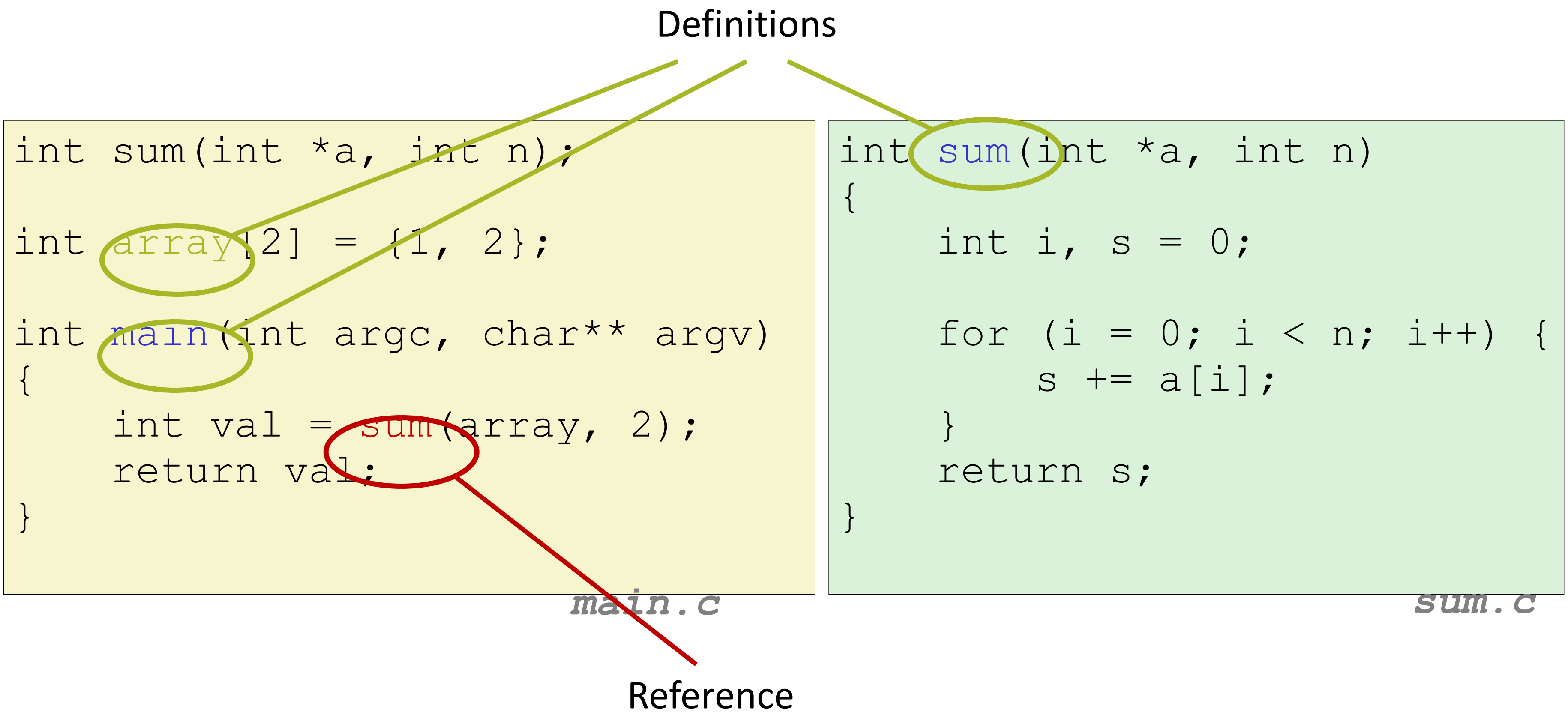
What Do Linkers Do?

Step 1: Symbol resolution（符号解析）

- Programs define and reference *symbols* (global variables and functions):
  - `void swap() {…} /* define symbol swap */`
  - `swap(); /* reference symbol swap */`
  - `int *xp = &x; /* define symbol xp, reference x */`

- Symbol definitions are stored in object file (by assembler) in symbol table（符号表）.
  - Symbol table is an array of `structs (gcc/include/elf.h)`
  - Each entry includes name, size, and location of symbol.

- During symbol resolution step, the linker associates each symbol reference with exactly one symbol definition.（明确符号引用到定义之间的关系）



Definitions

```
int sum(int *a, int n);

int array[2] = {1, 2};

int main(int argc, char** argv)
{
    int val = sum(array, 2);
    return val;
}
```

main.c

```
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

Reference

What do linkers do? (cont)

Step 2: Relocation （重定位）

- Merges separate code and data sections into single sections （合并代码段）
- Relocates symbols from their relative locations in the .o files to their final absolute memory locations in the executable. （符号重定位）
- Updates all references to these symbols to reflect their new positions. （更新引用）

Let’s look at these two steps in more detail….

- Relocatable object file (`.o` file)（可重定位目标文件）

  Contains code and data in a form that can be combined with other relocatable object files to form executable object file.

  Each `.o` file is produced from exactly one source (`.c`) file

- Executable object file (`a.out` file)（可执行文件）

  Contains code and data in a form that can be copied directly into memory and then executed.

- Shared object file (`.so` file)（共享目标文件）

  Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.

  Called *Dynamic Link Libraries* (DLLs) by Windows

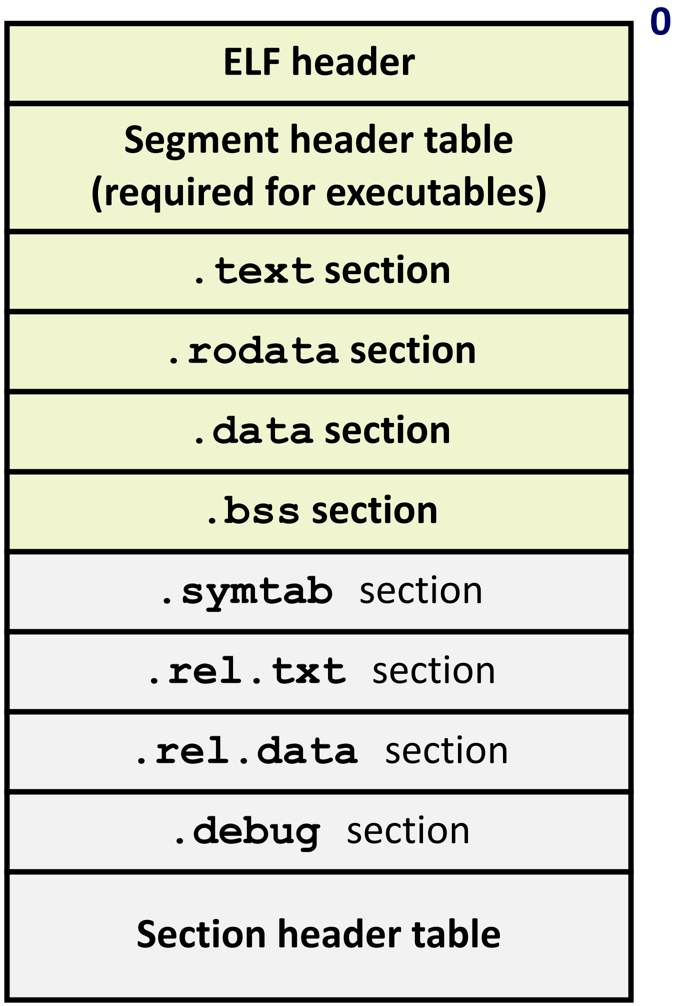
Standard binary format for object files

One unified format for

- Relocatable object files (`.o`),
- Executable object files `(a.out`)
- Shared object files (`.so`)

Generic name: ELF binaries

ELF Object File Format



0

ELF header

Segment header table
(required for executables)

.text section

.rodata section

.data section

.bss section

.symtab section

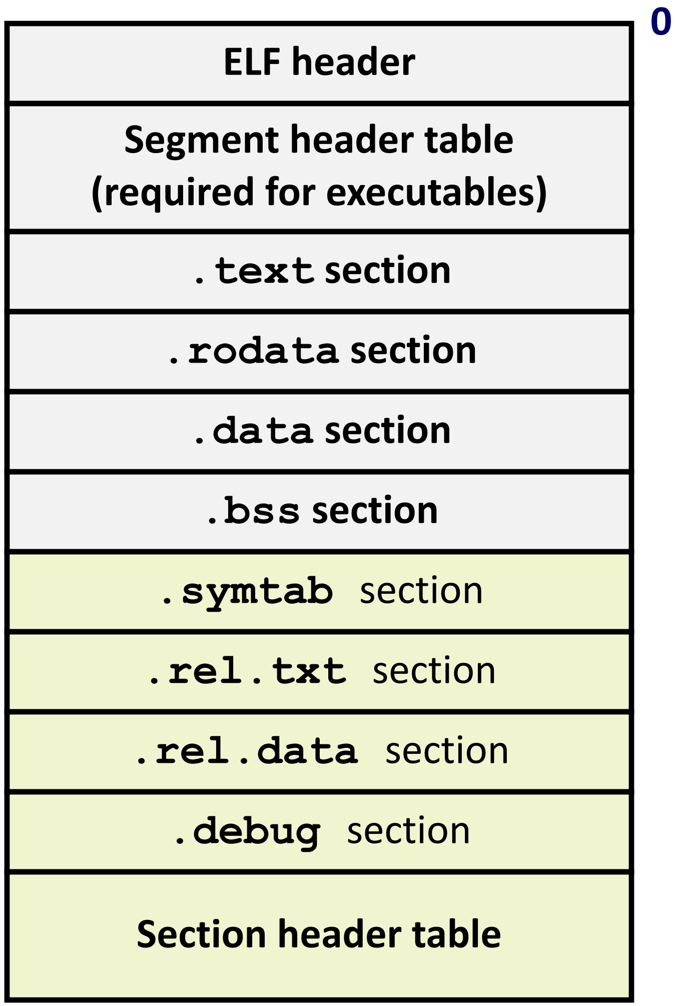
.rel.txt section

.rel.data section

.debug section

Section header table

ELF Object File Format



0

ELF header

Segment header table
(required for executables)

`.text` section

`.rodata` section

`.data` section

`.bss` section

`.symtab` section

`.rel.txt` section

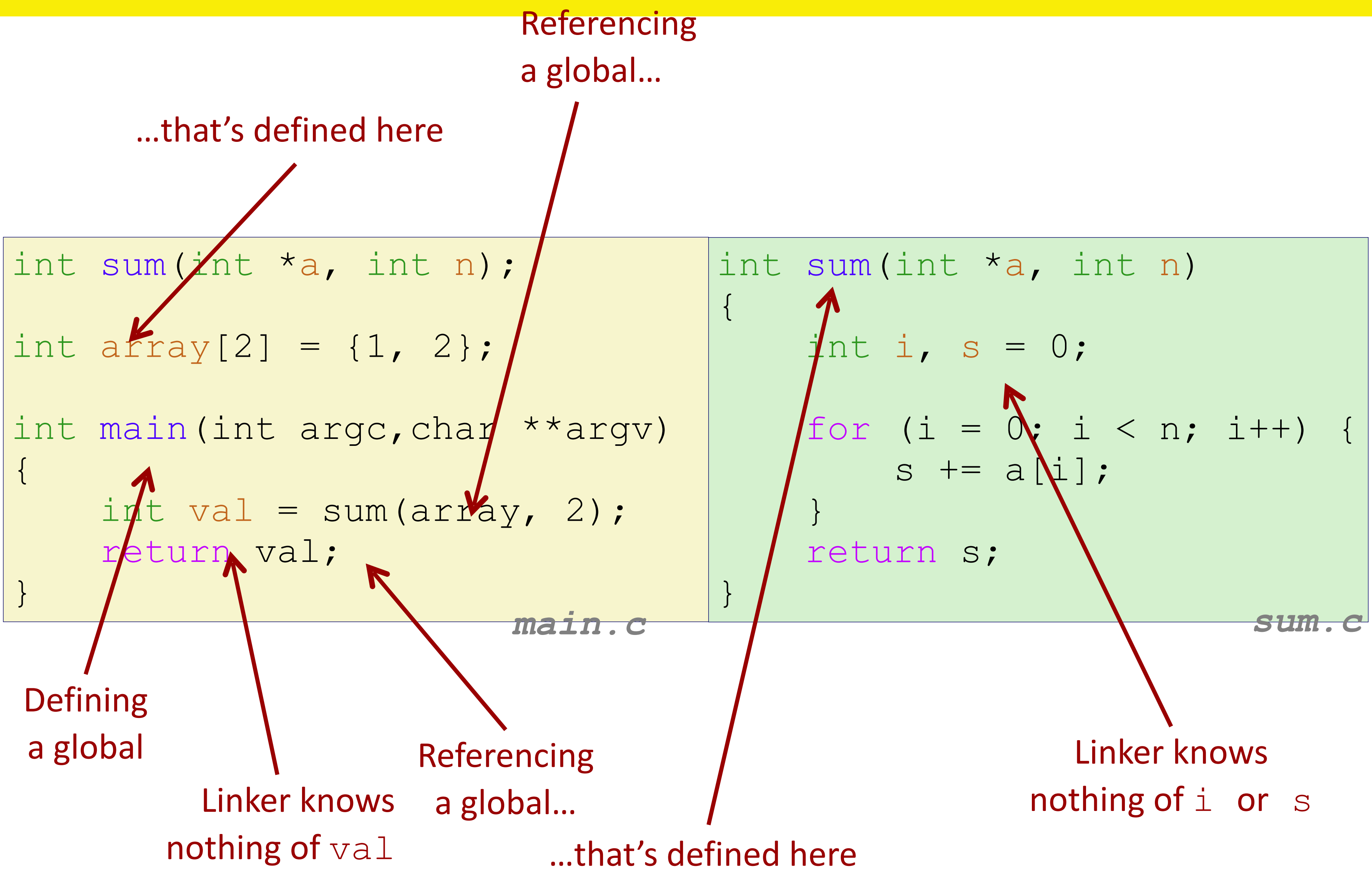
`.rel.data` section

`.debug` section

Section header table

Linker Symbols

Step 1: Symbol Resolution



Referencing a global…

…that's defined here

```
int sum(int *a, int n);

int array[2] = {1, 2};

int main(int argc,char **argv)
{
    int val = sum(array, 2);
    return val;
}
```

main.c

```
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

sum.c

Defining a global

Linker knows nothing of `val`

Referencing a global…

…that's defined here

Linker knows nothing of `i` or `s`

symbols.**c**:

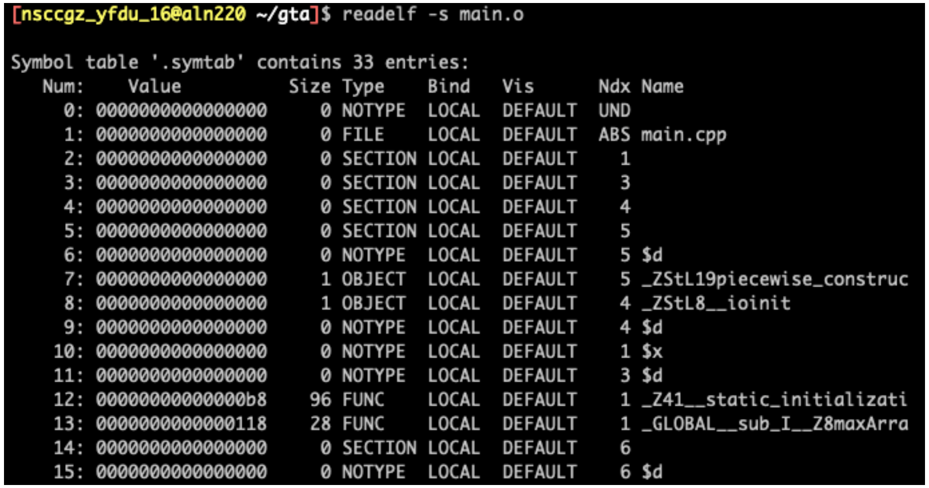
```
int incr = 1;
static int foo(int a) {
  int b = a + incr;
  return b;
}

int main(int argc,
           char* argv[]) {
  printf("%d\n", foo(5));
  return 0;
}
```

Names:

- `incr`
- `foo`
- `a`
- `argc`
- `argv`
- `b`
- `main`
- `printf`
- `"%d\n"`

Can find this with `readelf`:

```
linux> readelf -s symbols.o
```



```
[nsccgz_yfdu_16@aln220 ~/gta]$ readelf -s main.o

Symbol table '.symtab' contains 33 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS main.cpp
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    4
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     6: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT    5 $d
     7: 0000000000000000     1 OBJECT  LOCAL  DEFAULT    5 _ZStL19piecewise_construc
     8: 0000000000000000     1 OBJECT  LOCAL  DEFAULT    4 _ZStL8__ioinit
     9: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT    4 $d
    10: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT    1 $x
    11: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT    3 $d
    12: 00000000000000b8    96 FUNC    LOCAL  DEFAULT    1 _Z41__static_initializati
    13: 0000000000000118    28 FUNC    LOCAL  DEFAULT    1 _GLOBAL__sub_I__Z8maxArra
    14: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
    15: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT    6 $d
```

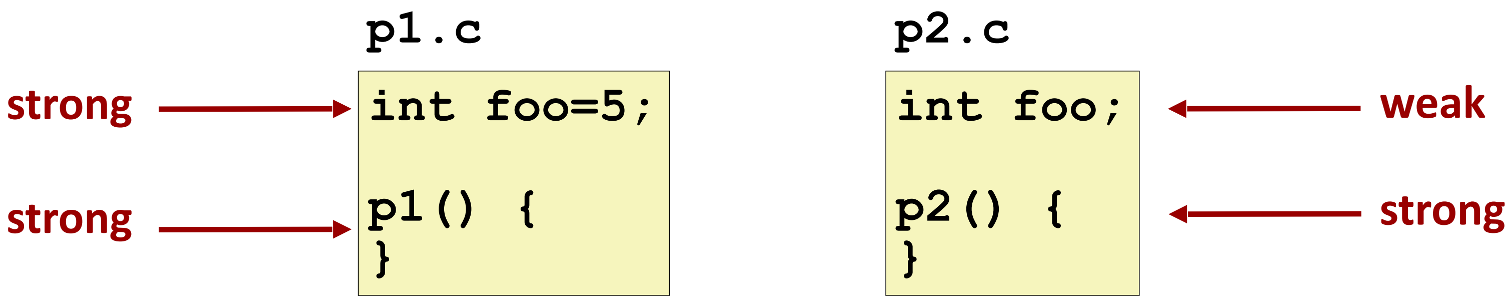
```
static int x = 15;

int f() {
    static int x = 17;
    return x++;
}

int g() {
    static int x = 19;
    return x += 14;
}

int h() {
    return x += 27;
}
```

static-local.c

Compiler allocates space in `.data` for each definition of `x`

Creates local symbols in the symbol table with unique names, e.g., `x`, `x.1721` and `x.1724`.

How Linker Resolves Duplicate Symbol Definitions



p1.c

```
int foo=5;

p1() {
}
```

p2.c

```
int foo;

p2() {
}
```

Linker’s Symbol Rules

Linker Puzzles

```
int x;
p1() {}
```

```
p1() {}
```

Link time error: two strong symbols (`p1`)

```
int x;
p1() {}
```

```
int x;
p2() {}
```

References to `x` will refer to the same uninitialized int. Is this what you really want?

```
int x;
int y;
p1() {}
```

```
double x;
p2() {}
```

Writes to `x` in `p2` might overwrite `y`! Evil!

```
int x=7;
int y=5;
p1() {}
```

```
double x;
p2() {}
```

Writes to `x` in `p2` might overwrite `y`! Nasty!

```
int x=7;
p1() {}
```

```
int x;
p2() {}
```

References to `x` will refer to the same initialized variable.

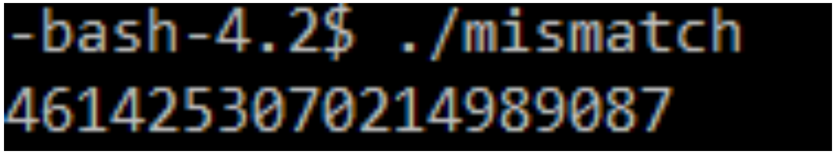
Important: Linker does not do type checking.

```c
long int x;   /* Weak symbol */

int main(int argc,
         char *argv[]) {
    printf("%ld\n", x);
    return 0;
}
```

mismatch-main.c

```c
/* Global strong symbol */
double x = 3.14;
```

mismatch-variable.c



```
-bash-4.2$ ./mismatch
4614253070214989087
```

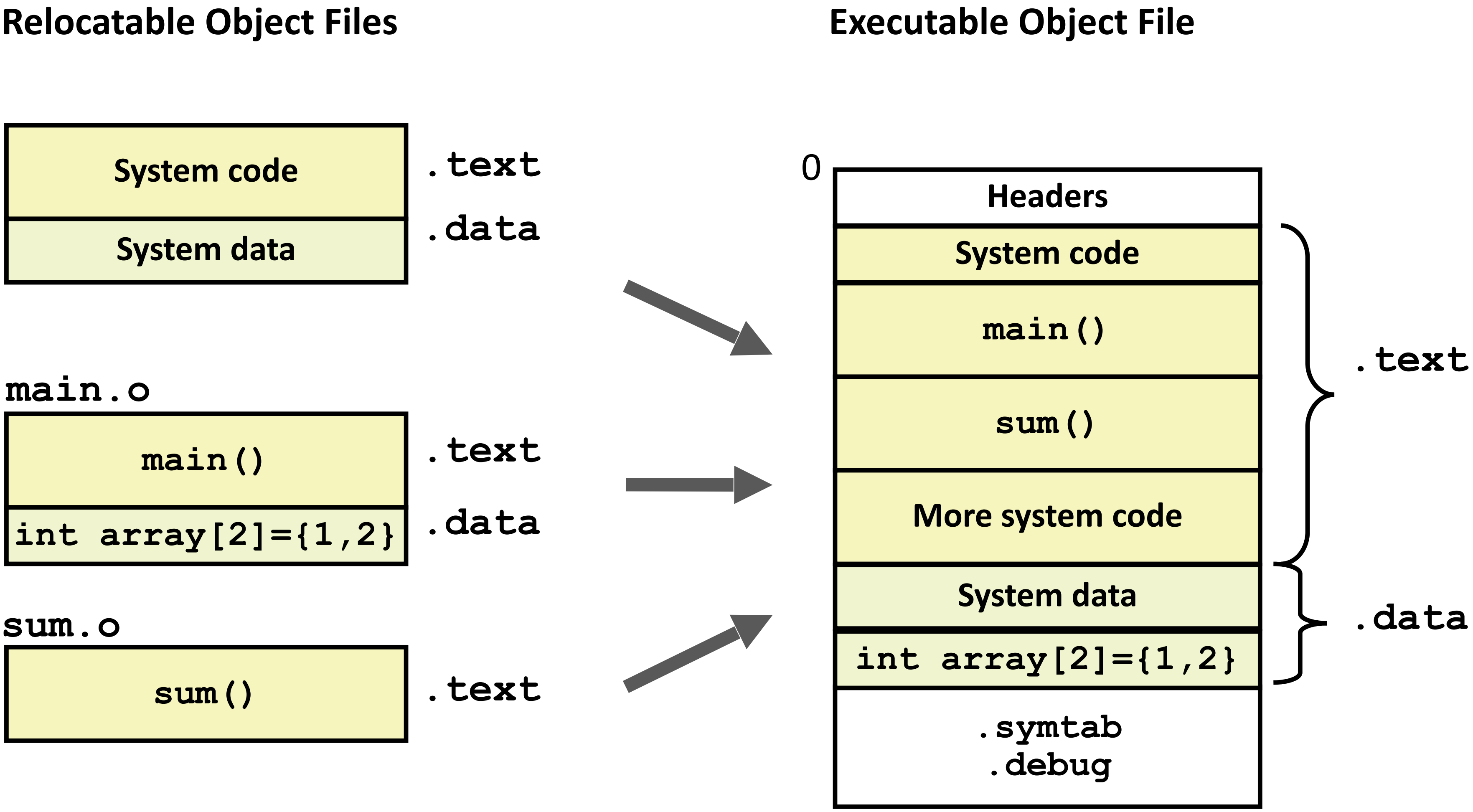
- Avoid if you can

- Otherwise
  - Use **`static`** if you can
  - Initialize if you define a global variable
  - Use **`extern`** if you reference an external global variable

    Treated as weak symbol

    But also causes linker error if not defined in some file

Step 2: Relocation



Relocation Entries

```
int array[2] = {1, 2};

int main(int argc, char**
argv)
{
    int val = sum(array, 2);
    return val;
}
```

main.c

```
0000000000000000 <main>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   be 02 00 00 00          mov    $0x2,%esi
   9:   bf 00 00 00 00          mov    $0x0,%edi      # %edi = &array
                        a: R_X86_64_32 array          # Relocation entry

   e:   e8 00 00 00 00          callq  13 <main+0x13> # sum()
                        f: R_X86_64_PC32 sum-0x4      # Relocation entry
  13:   48 83 c4 08             add    $0x8,%rsp
  17:   c3                      retq
```

main.o

Source: `objdump -r -d main.o`

Relocated .text section

```
00000000004004d0 <main>:
  4004d0:       48 83 ec 08         sub    $0x8,%rsp
  4004d4:       be 02 00 00 00      mov    $0x2,%esi
  4004d9:       bf 18 10 60 00      mov    $0x601018,%edi  # %edi = &array
  4004de:       e8 05 00 00 00      callq  4004e8 <sum>    # sum()
  4004e3:       48 83 c4 08         add    $0x8,%rsp
  4004e7:       c3                  retq

00000000004004e8 <sum>:
  4004e8:       b8 00 00 00 00            mov    $0x0,%eax
  4004ed:       ba 00 00 00 00            mov    $0x0,%edx
  4004f2:       eb 09                     jmp    4004fd <sum+0x15>
  4004f4:       48 63 ca                  movslq %edx,%rcx
  4004f7:       03 04 8f                  add    (%rdi,%rcx,4),%eax
  4004fa:       83 c2 01                  add    $0x1,%edx
  4004fd:       39 f2                     cmp    %esi,%edx
  4004ff:       7c f3                     jl     4004f4 <sum+0xc>
  400501:       f3 c3                     repz retq
```

`callq` instruction uses PC-relative addressing for sum():

`0x4004e8` = `0x4004e3` + `0x5`

Source: `objdump -d prog`

中山大學 SUN YAT-SEN UNIVERSITY 国家超级计算广州中心 NATIONAL SUPERCOMPUTER CENTER IN GUANGZHOU

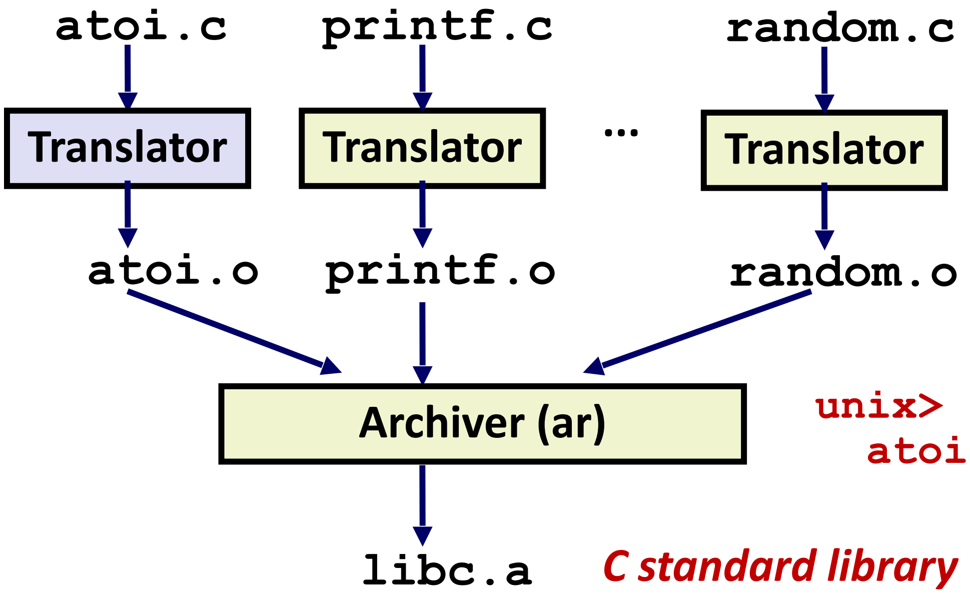
Linking with Static Libraries

Static libraries (.a archive files)

Concatenate related relocatable object files into a single file with an index (called an *archive*).

Enhance linker so that it tries to resolve unresolved external references by looking for the symbols in one or more archives.

If an archive member file resolves reference, link it into the executable.



```
unix> ar rs libc.a \
   atoi.o printf.o … random.o
```

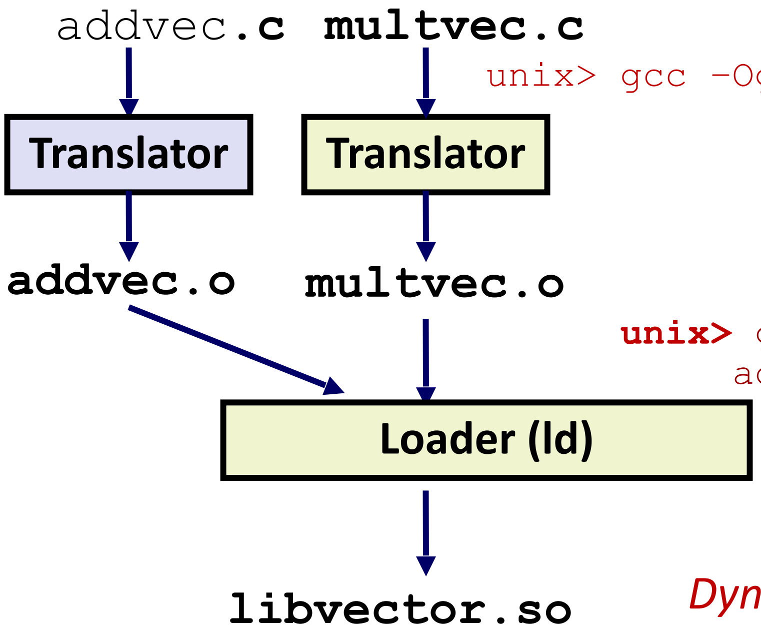
C standard library

Linking with Shared Libraries

shared libraries

Object files that contain code and data that are loaded and linked into an application *dynamically,* at either *load-time* or *run-time*

Also called: dynamic link libraries, DLLs, `.so` files



```
unix> gcc -Og -c addvec.c multvec.c -fpic
```

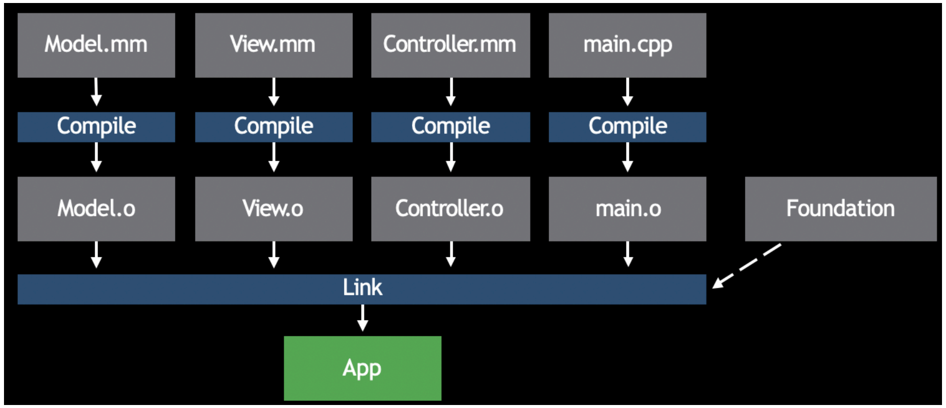
```
unix> gcc -shared -o libvector.so \
      addvec.o multvec.o
```

Dynamic vector library

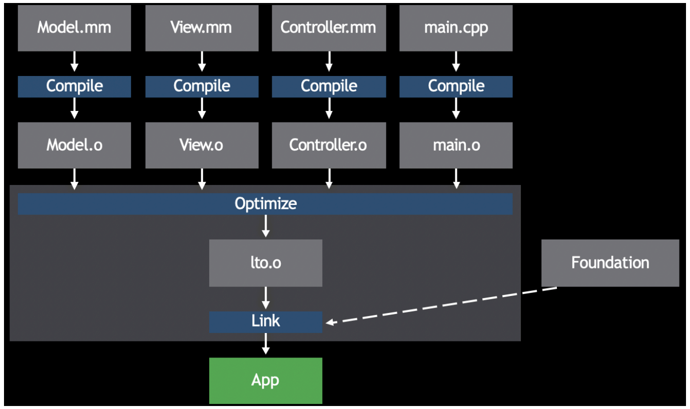
Can we do optimizations when linking?

Link-Time Optimization

Traditional Compilation Model



LTO Compilation Model



Maximize runtime performance by optimizing at link-time

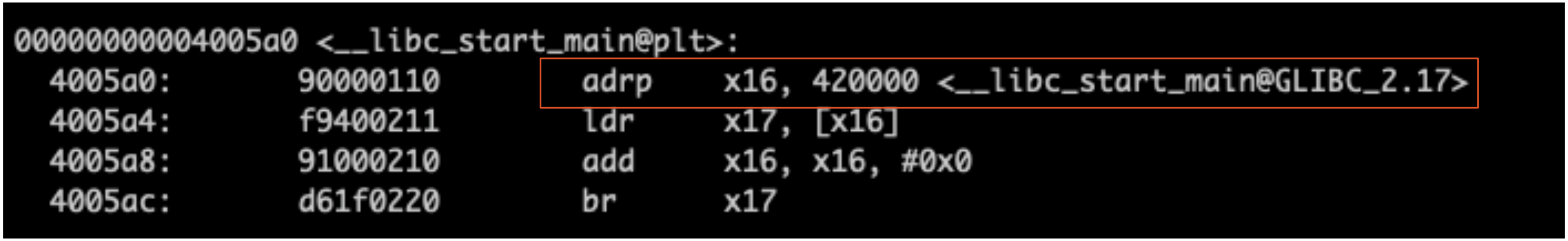
- Inline functions across source files
- Remove dead code
- Enable powerful whole program optimizations

Apple uses LTO extensively internally

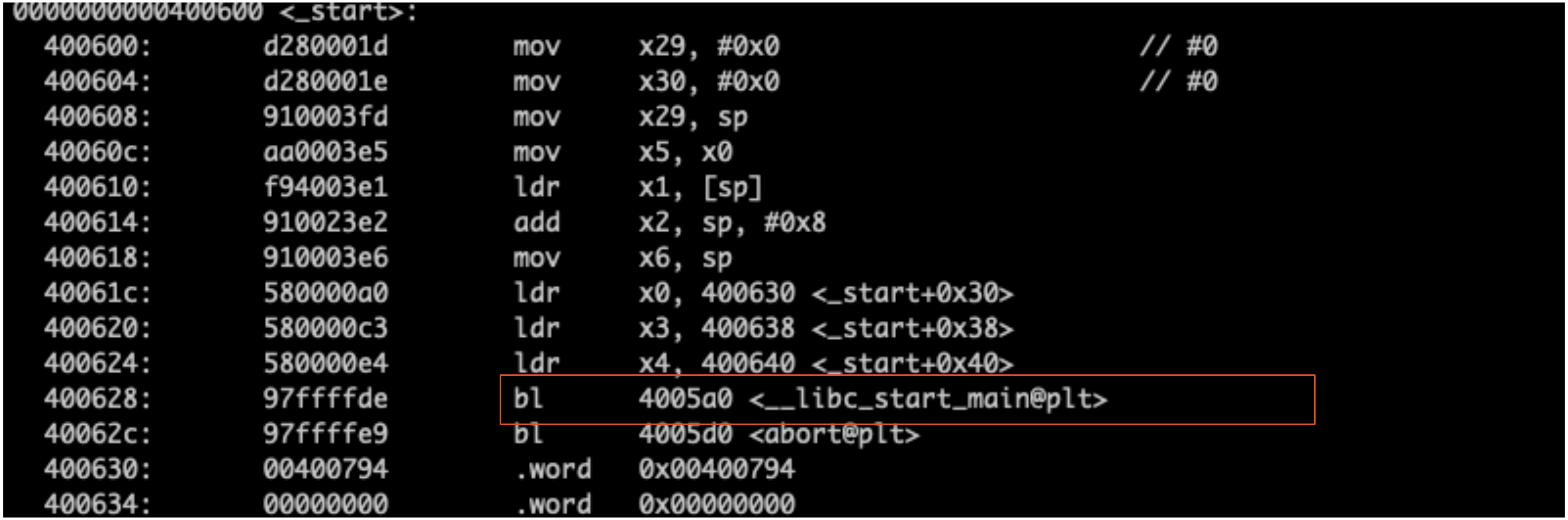
- Typically 10% faster than executables from regular Release builds
- Multiplies with Profile Guided Optimization (PGO)
- Reduces code size when optimizing for size

From Developer Tools #WWDC16 - Apple

How does a program run and can optimize more?



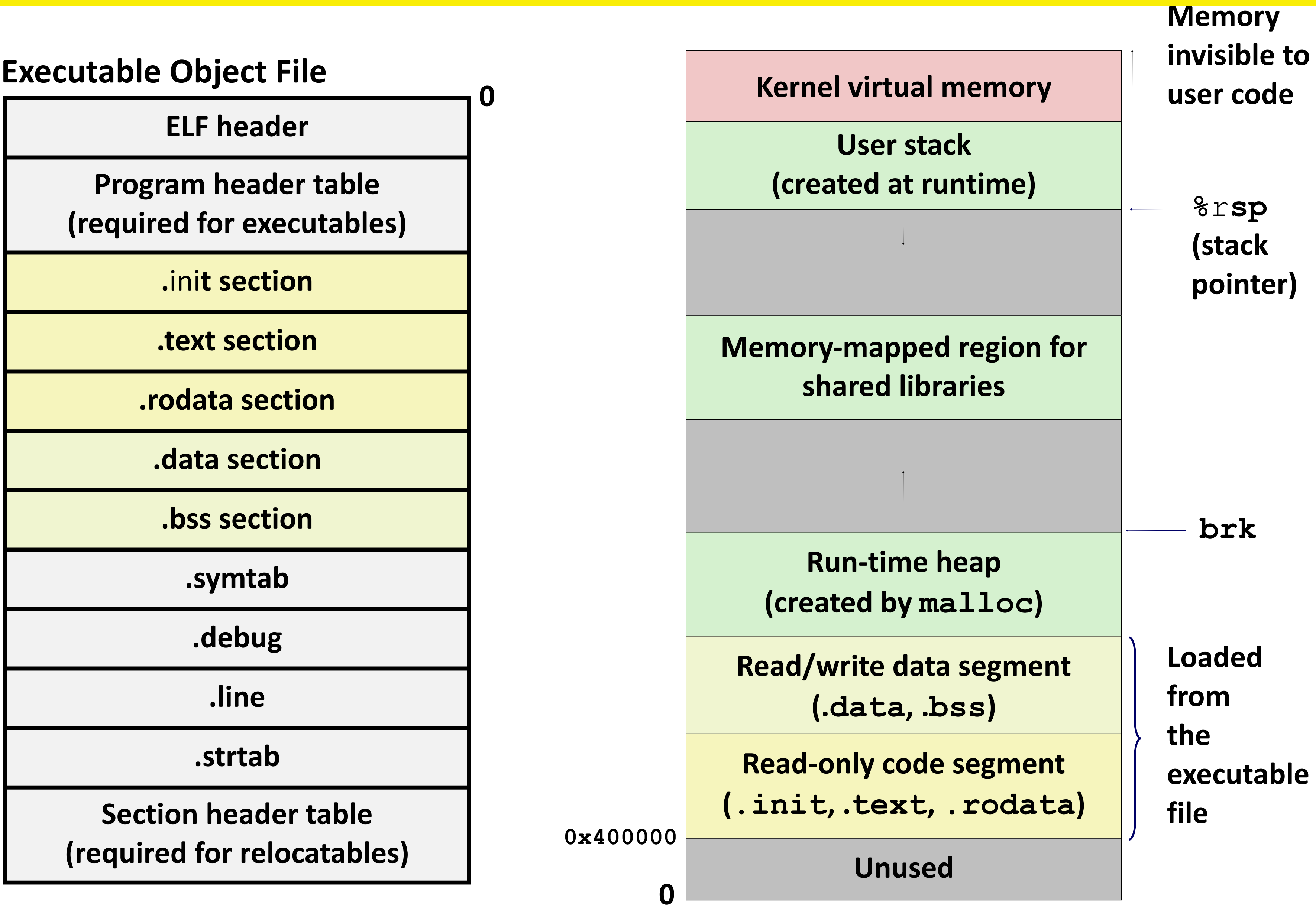
```
00000000004005a0 <__libc_start_main@plt>:
  4005a0:       90000110        adrp    x16, 420000 <__libc_start_main@GLIBC_2.17>
  4005a4:       f9400211        ldr     x17, [x16]
  4005a8:       91000210        add     x16, x16, #0x0
  4005ac:       d61f0220        br      x17
```



```
0000000000400600 <_start>:
  400600:       d280001d        mov     x29, #0x0                       // #0
  400604:       d280001e        mov     x30, #0x0                       // #0
  400608:       910003fd        mov     x29, sp
  40060c:       aa0003e5        mov     x5, x0
  400610:       f94003e1        ldr     x1, [sp]
  400614:       910023e2        add     x2, sp, #0x8
  400618:       910003e6        mov     x6, sp
  40061c:       580000a0        ldr     x0, 400630 <_start+0x30>
  400620:       580000c3        ldr     x3, 400638 <_start+0x38>
  400624:       580000e4        ldr     x4, 400640 <_start+0x40>
  400628:       97fffdde        bl      4005a0 <__libc_start_main@plt>
  40062c:       97ffffe9        bl      4005d0 <abort@plt>
  400630:       00400794        .word   0x00400794
  400634:       00000000        .word   0x00000000
```

https://www.gnu.org/software/hurd/glibc/startup.html

Loading Executable Object Files



Executable Object File

0

ELF header

Program header table
(required for executables)

.init section

.text section

.rodata section

.data section

.bss section

.symtab

.debug

.line

.strtab

Section header table
(required for relocatables)

Kernel virtual memory

Memory invisible to user code

User stack
(created at runtime)

%rsp
(stack pointer)

Memory-mapped region for shared libraries

brk

Run-time heap
(created by malloc)

Read/write data segment
(.data, .bss)

Read-only code segment
(.init, .text, .rodata)

Loaded from the executable file

0x400000

Unused

0

Can optimize more?

Binary Optimizer

- No need to link sample-based profile data to source code or IR
- Can optimize 3rd -party libraries without source code
- Has “whole-program” view
- Some optimizations could only be done to a binary

BOLT: a practical binary optimizer for data centers and beyond. CGO 2019
Using LLVM for optimized lightweight binary re-writing at runtime. PDPSW 2017

Thank you,

and we look forward to welcoming you to NSCC-GZ.

What happens after compiling?
葛天傲
getao3@mail2.sysu.edu.cn
DCS290, 04/22/2021

Backup

Developer Tools

What’s New in LLVM Session 405

#WWDC16 - Apple

Developer Tools

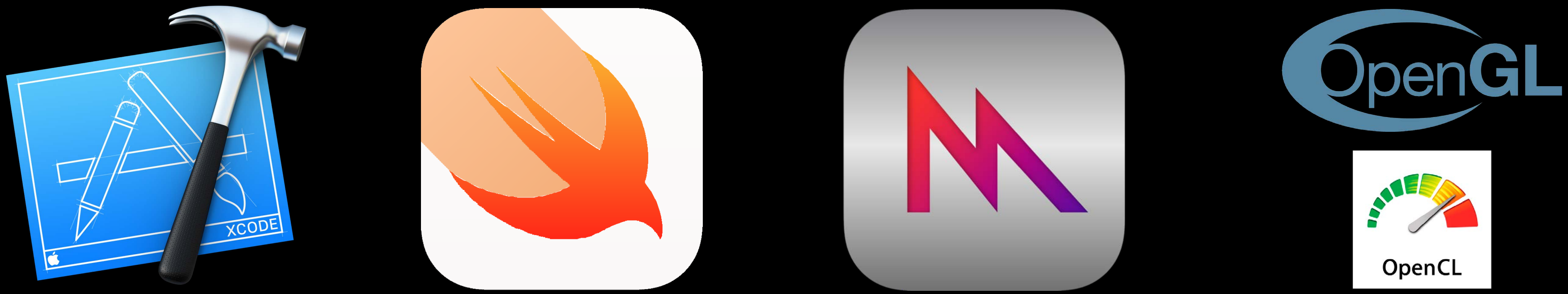
#WWDC16

What's New in LLVM

Session 405

Alex Rosenberg Final Boss Level, Compilers and Stuff
Duncan Exon Smith Manager, Clang Frontend
Gerolf Hoflehner Manager, LLVM Backend

LLVM is Everywhere



What is Link-Time Optimization (LTO)?

Maximize runtime performance by optimizing at link-time

- Inline functions across source files
- Remove dead code
- Enable powerful whole program optimizations

Traditional Compilation Model

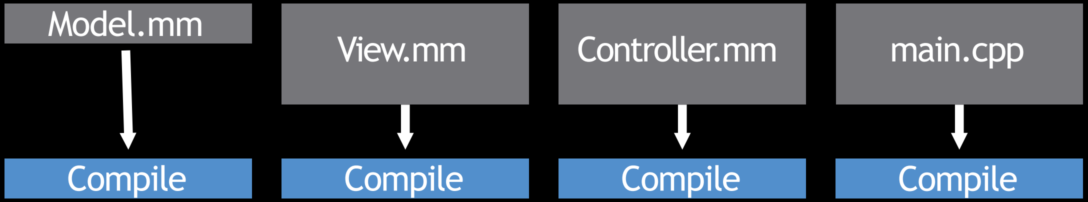
Model.mm

View.mm

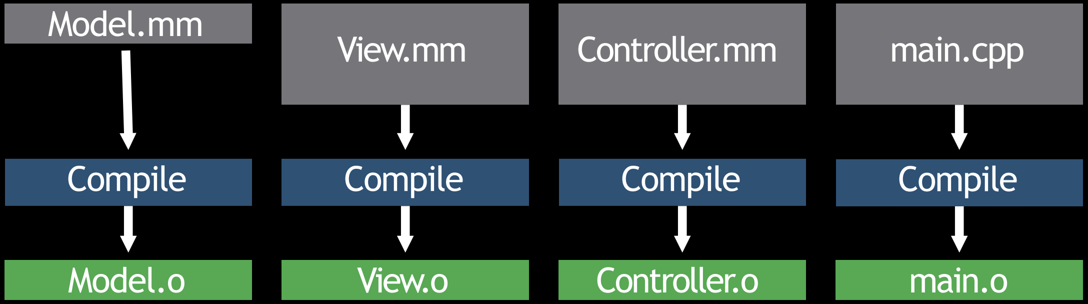
Controller.mm

main.cpp

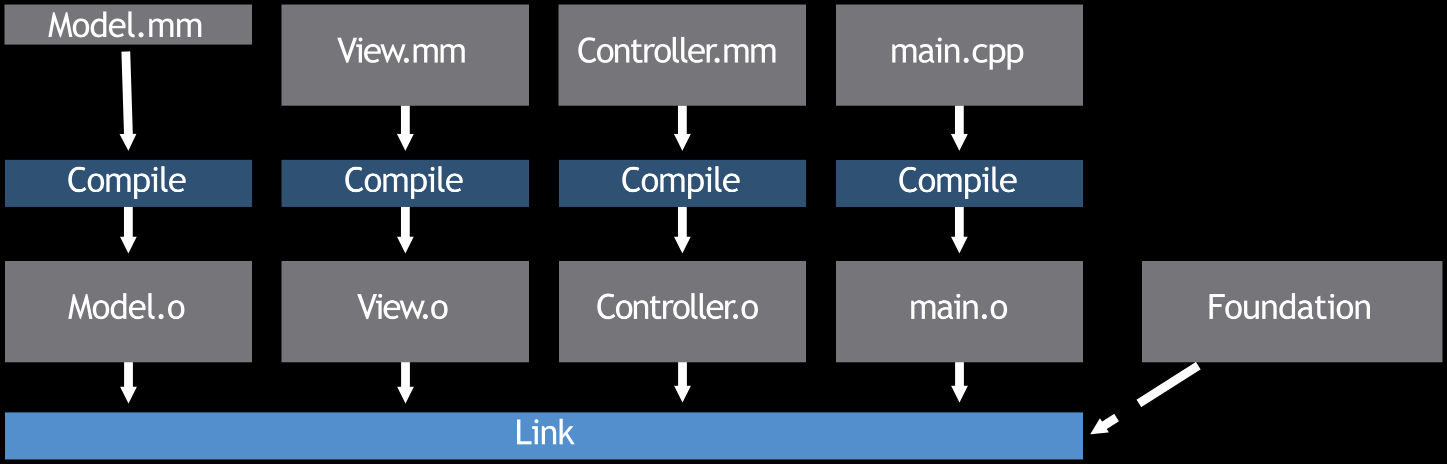
Traditional Compilation Model



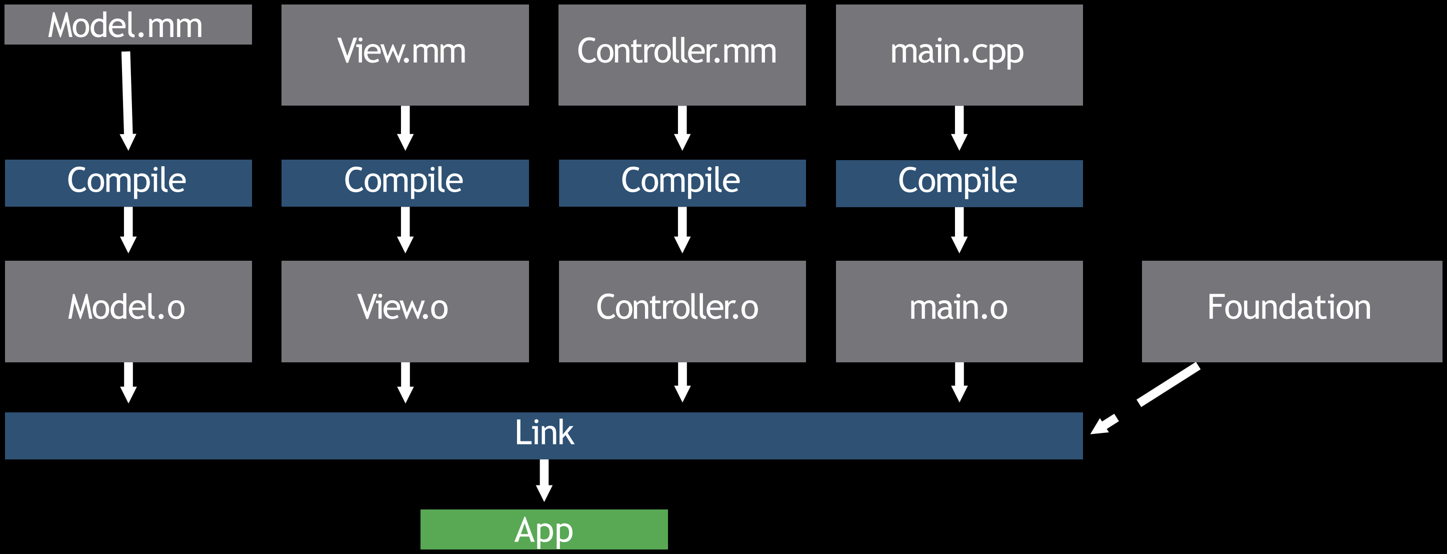
Traditional Compilation Model



Traditional Compilation Model



Traditional Compilation Model



LTO Compilation Model

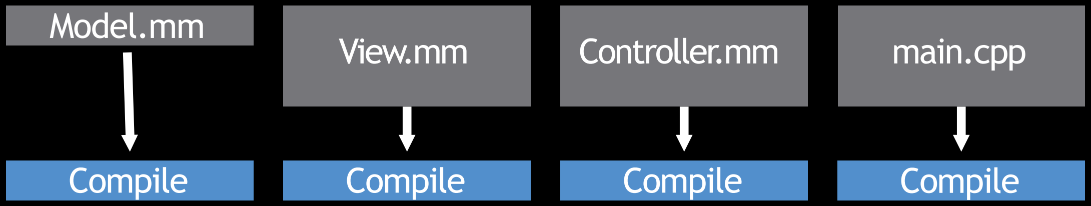
Model.mm

View.mm

Controller.mm

main.cpp

LTO Compilation Model



Model.mm

View.mm

Controller.mm

main.cpp

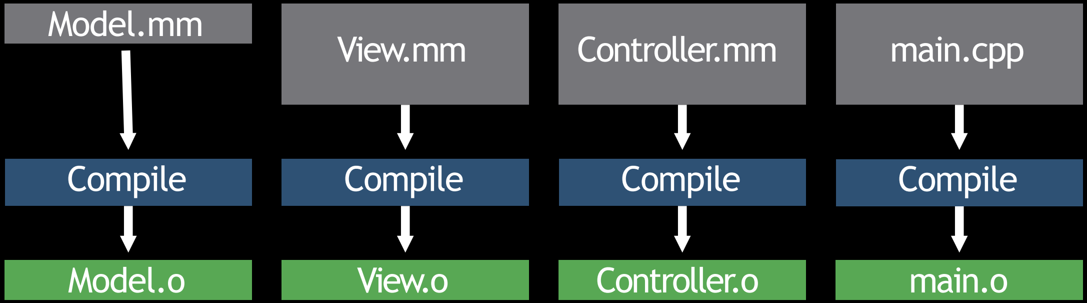
Compile

Compile

Compile

Compile

LTO Compilation Model



Model.mm

View.mm

Controller.mm

main.cpp

Compile

Compile

Compile

Compile

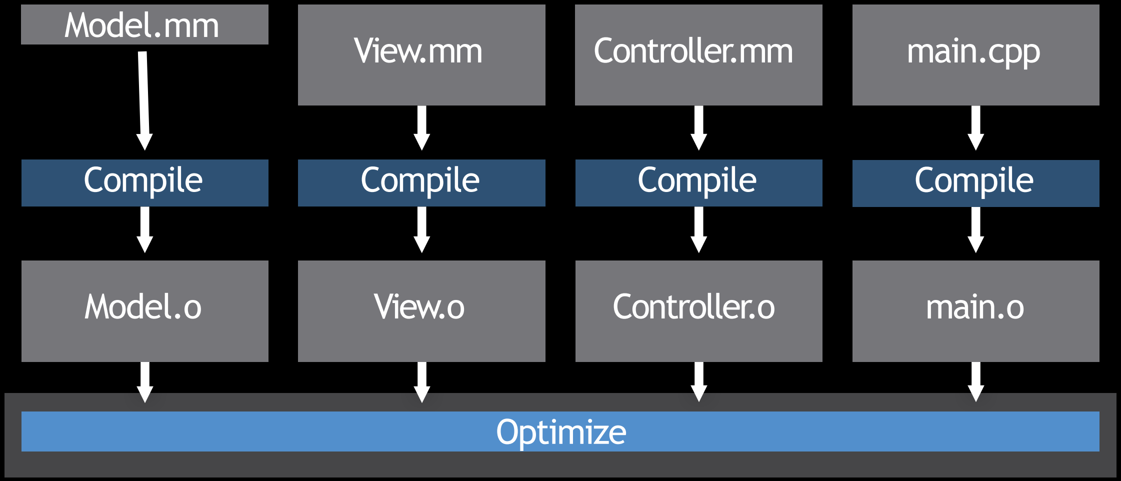
Model.o

View.o

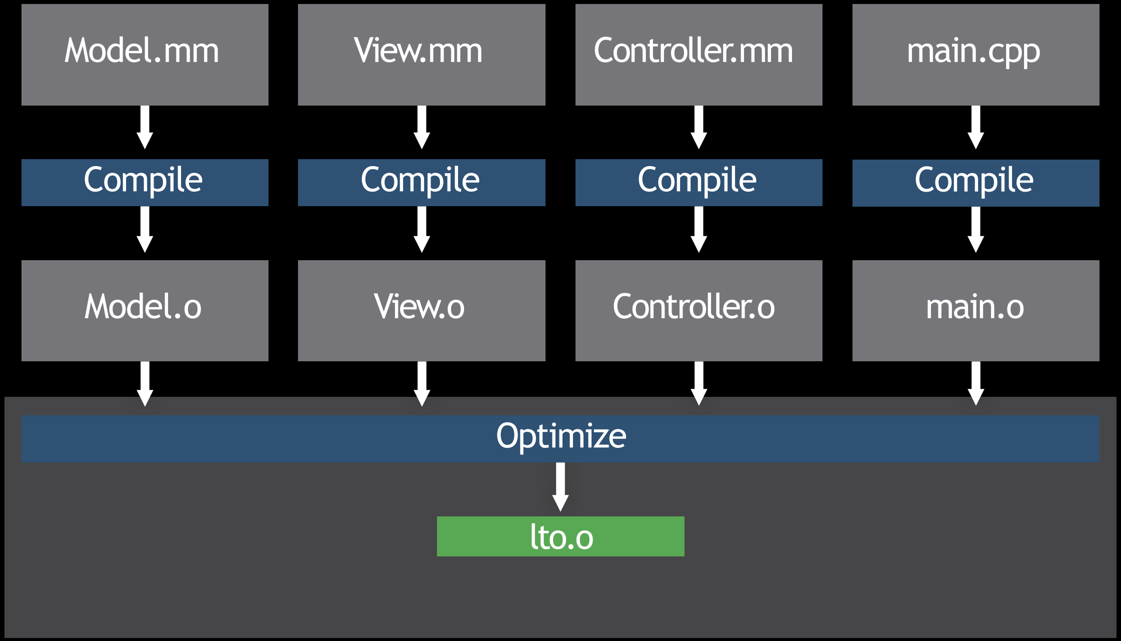
Controller.o

main.o

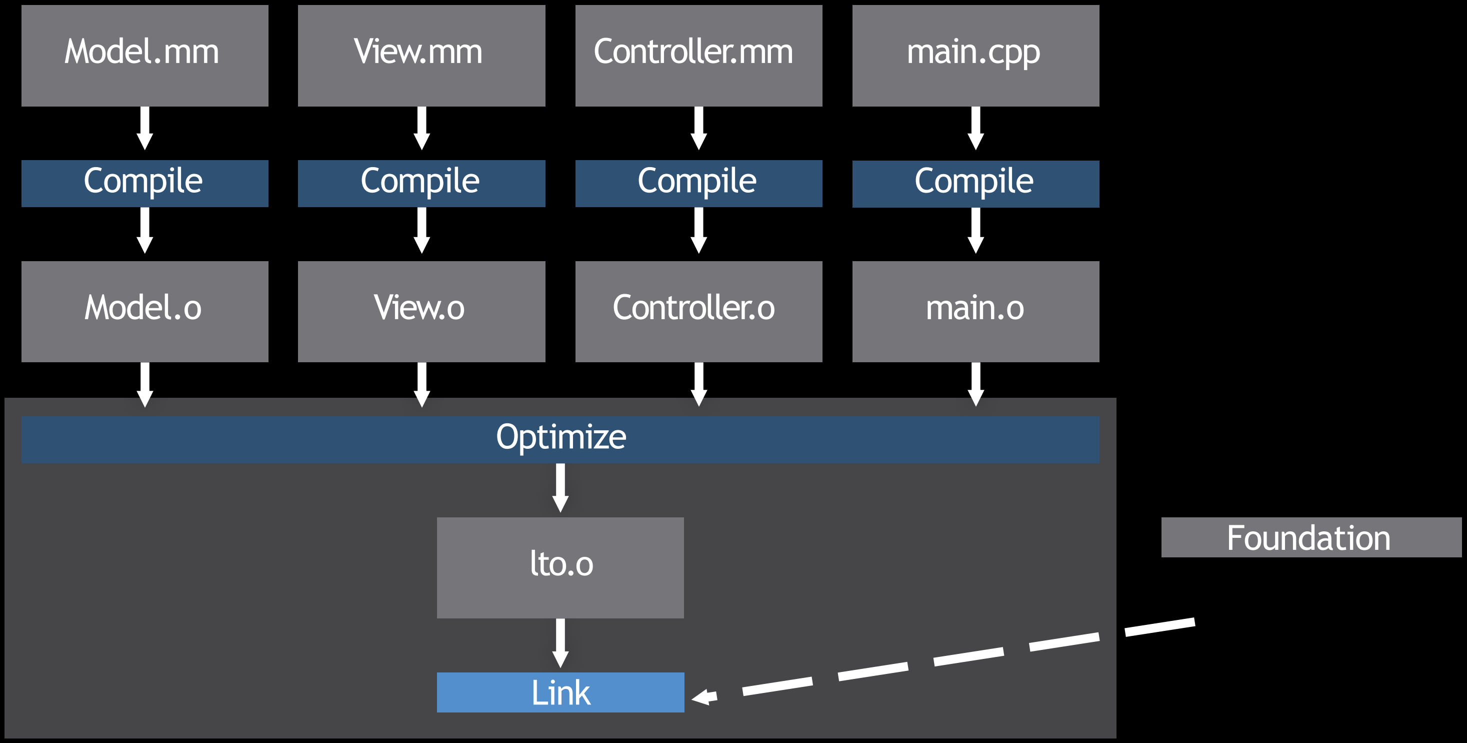
LTO Compilation Model



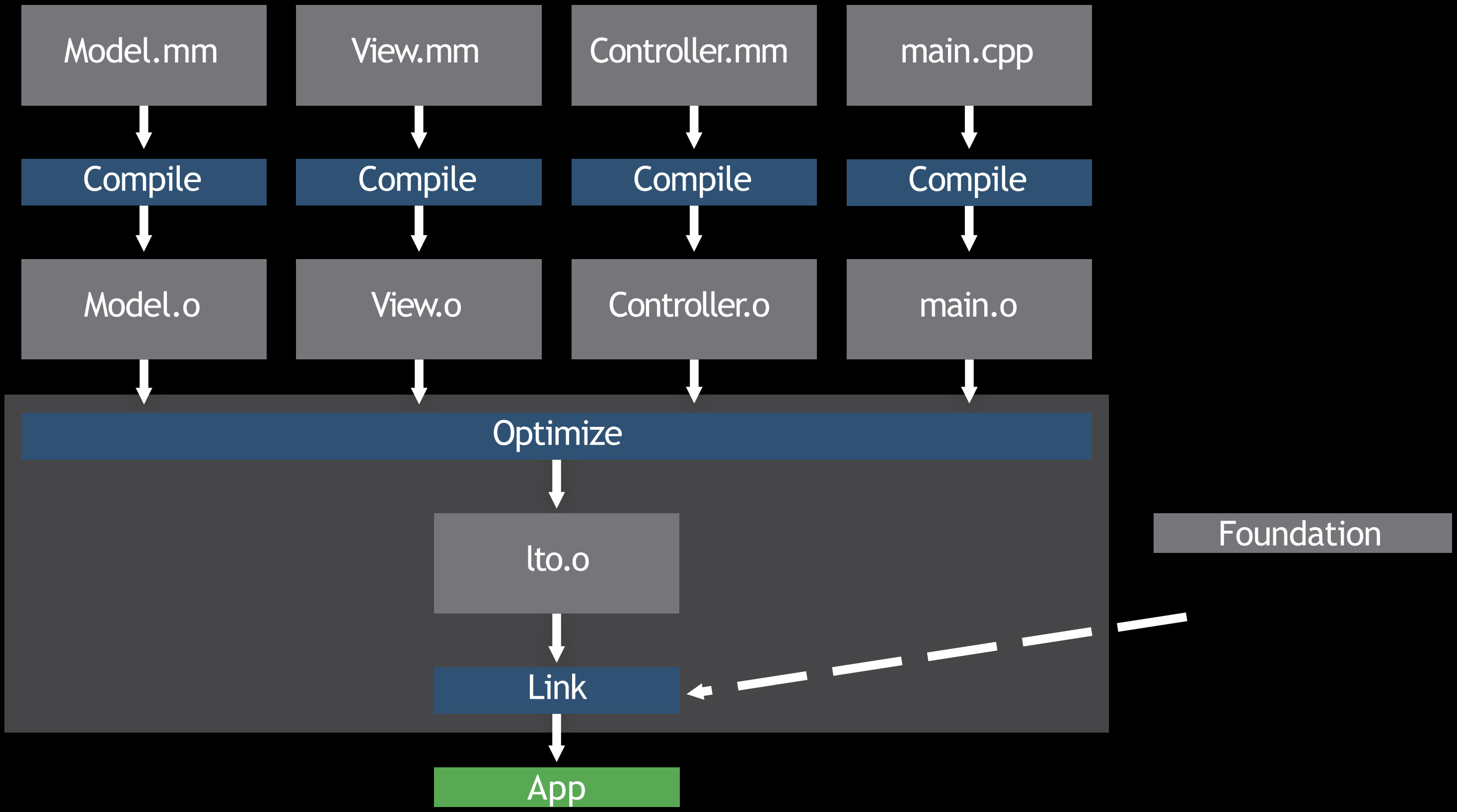
LTO Compilation Model



LTO Compilation Model



LTO Compilation Model



LTO Runtime Performance

Maximize performance with LTO

Apple uses LTO extensively internally

LTO Runtime Performance

Maximize performance with LTO

Apple uses LTO extensively internally

- Typically 10% faster than executables from regular Release builds

LTO Runtime Performance

Maximize performance with LTO

Apple uses LTO extensively internally

- Typically 10% faster than executables from regular Release builds
- Multiplies with Profile Guided Optimization (PGO)

LTO Runtime Performance

Maximize performance with LTO

Apple uses LTO extensively internally

- Typically 10% faster than executables from regular Release builds
- Multiplies with Profile Guided Optimization (PGO)
- Reduces code size when optimizing for size

LTO Compile Time Tradeoff

LTO trades compile time for runtime performance

LTO Compile Time Tradeoff

LTO trades compile time for runtime performance

- Large memory requirements

LTO Compile Time Tradeoff

LTO trades compile time for runtime performance

- Large memory requirements
- Optimizations are not done in parallel

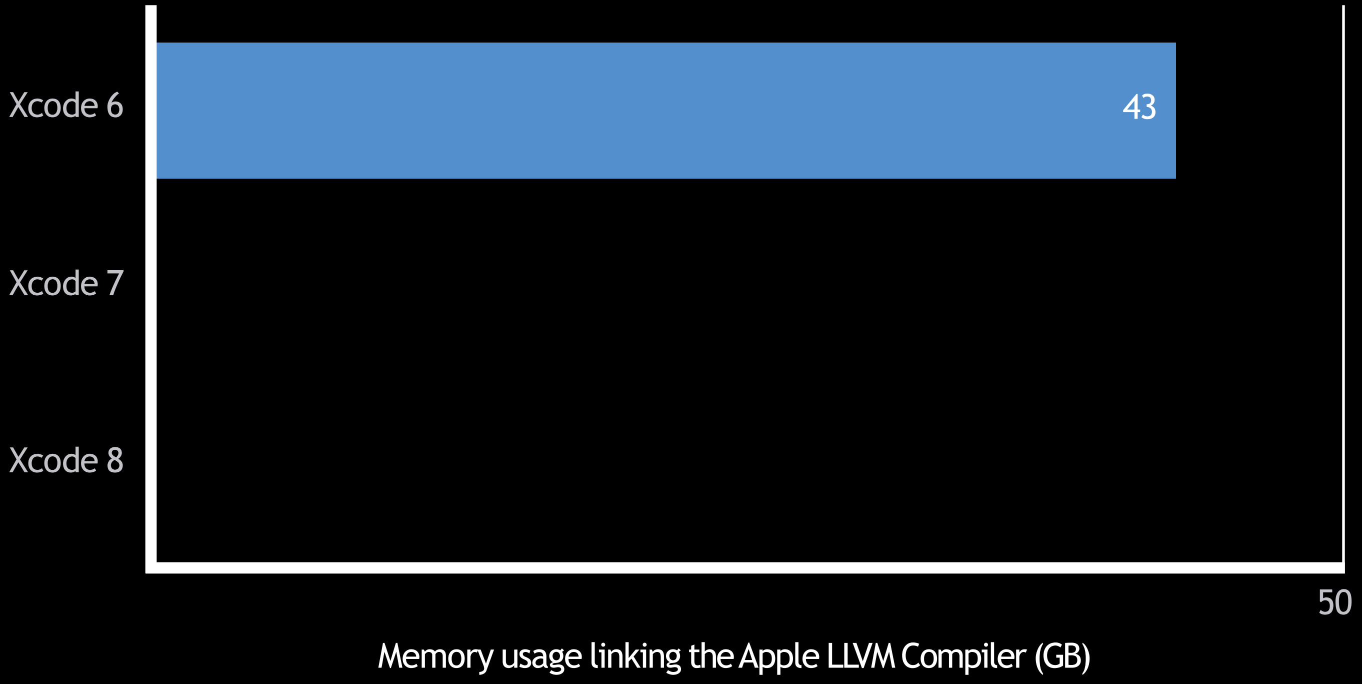
LTO Compile Time Tradeoff

LTO trades compile time for runtime performance

- Large memory requirements
- Optimizations are not done in parallel
- Incremental builds repeat all the work

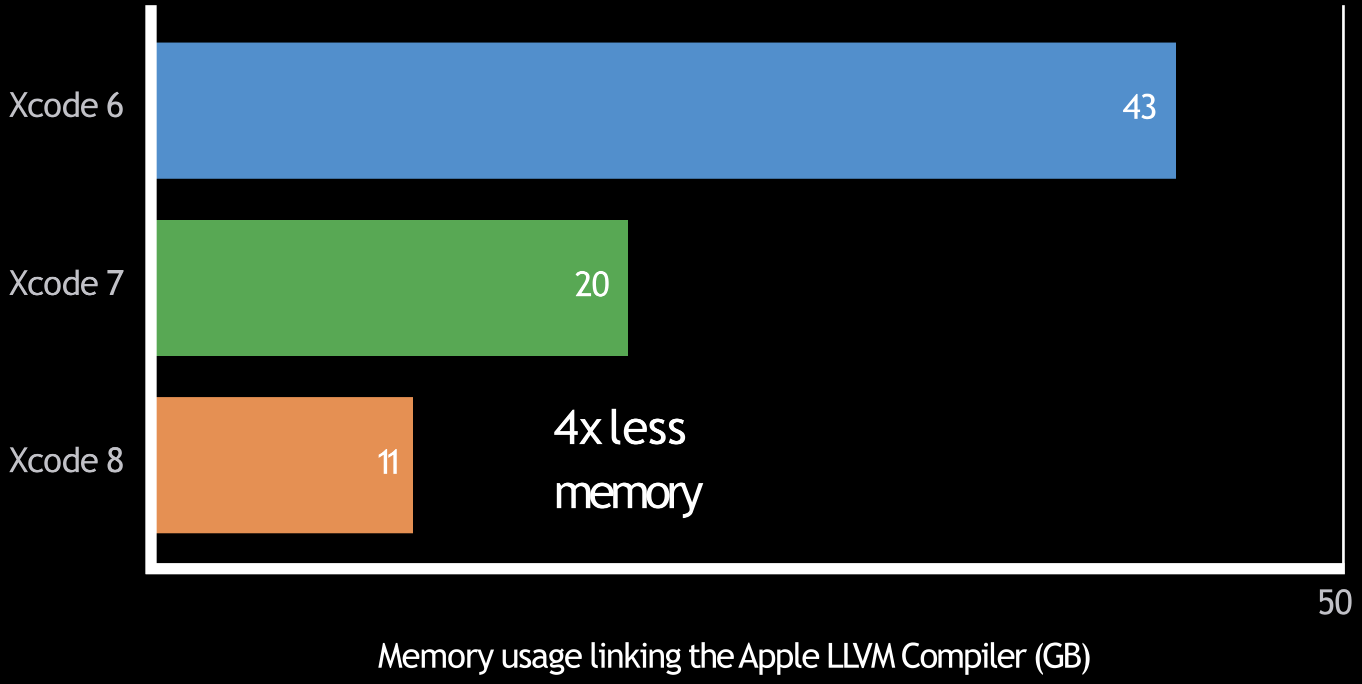
LTO Memory Usage—Full Debug Info

Large C++ project with -g



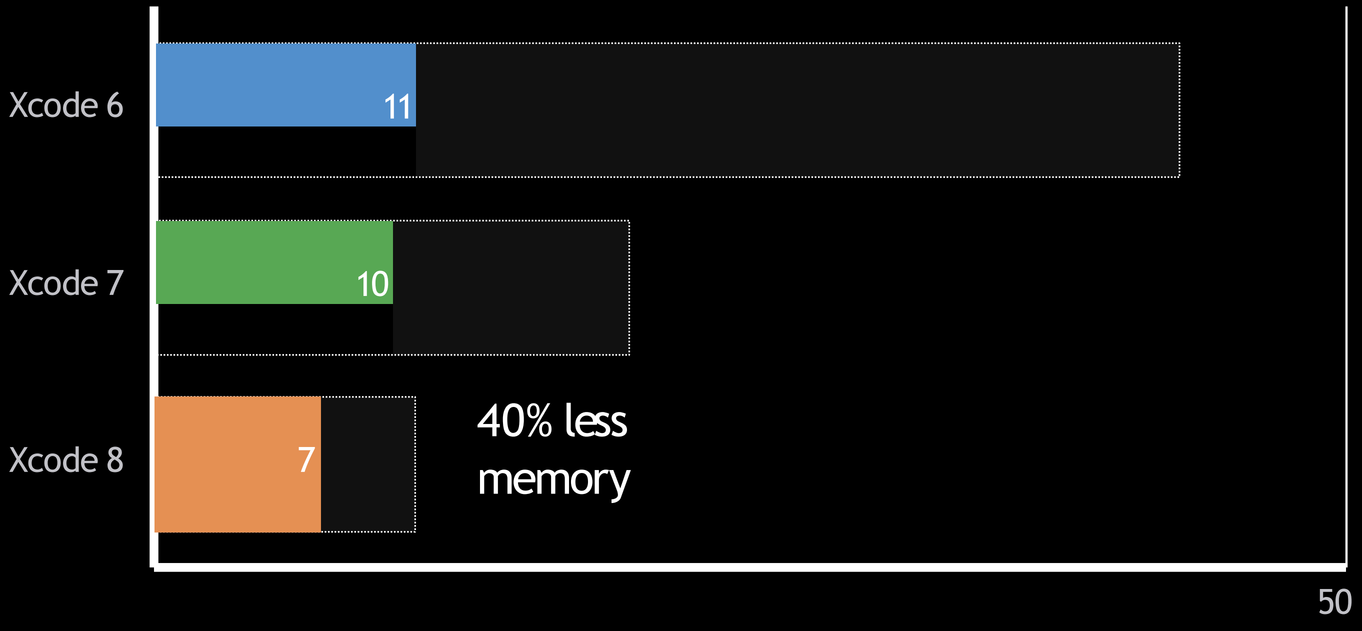
LTO Memory Usage—Full Debug Info

Large C++ project with -g



LTO Memory Usage — Line Tables Only

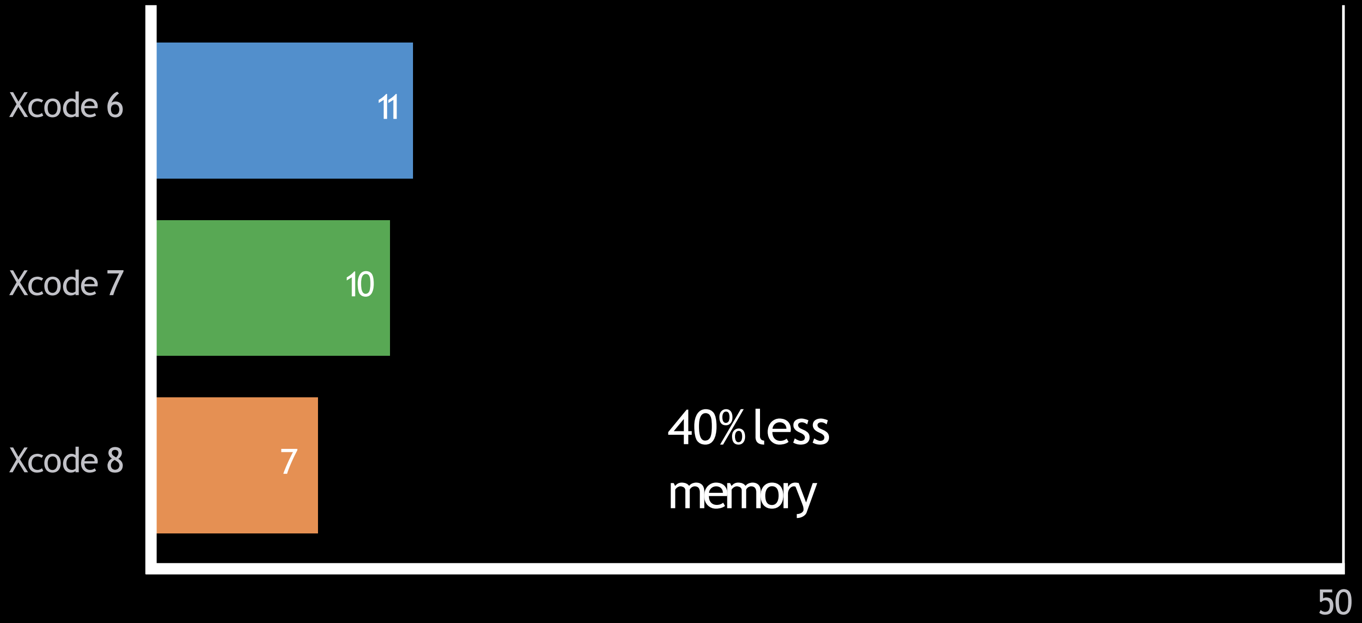
Large C++ project with `-gline-tables-only`



Memory usage linking the Apple LLVM Compiler (GB)

LTO Memory Usage — Line Tables Only

Large C++ project with `-gline-tables-only`



40% less memory

Memory usage linking the Apple LLVM Compiler (GB)

Incremental LTO

NEW

New model for link-time optimization that scales with your system

Incremental LTO

NEW

New model for link-time optimization that scales with your system

- Analysis and inlining without merging object files

Incremental LTO

NEW

New model for link-time optimization that scales with your system

- Analysis and inlining without merging object files
- Optimizations run in parallel

Incremental LTO

NEW

New model for link-time optimization that scales with your system

- Analysis and inlining without merging object files
- Optimizations run in parallel
- Linker cache for fast incremental builds

Incremental LTO Compilation Model

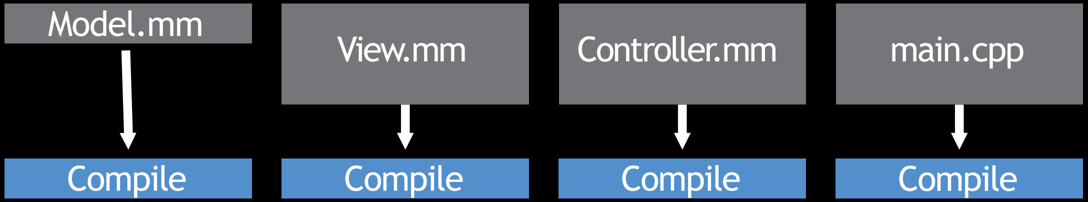
Model.mm

View.mm

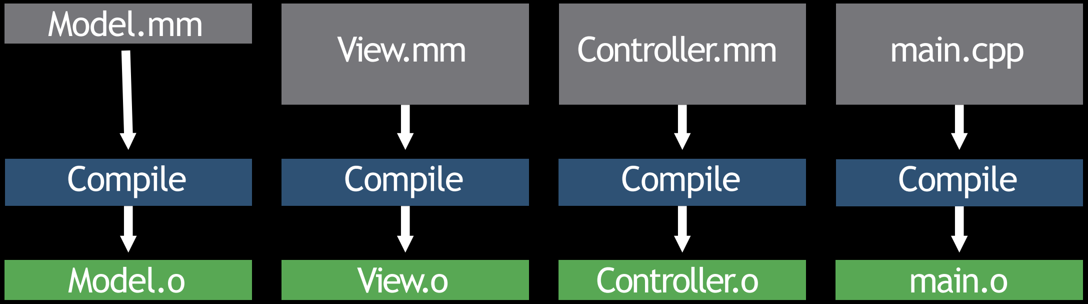
Controller.mm

main.cpp

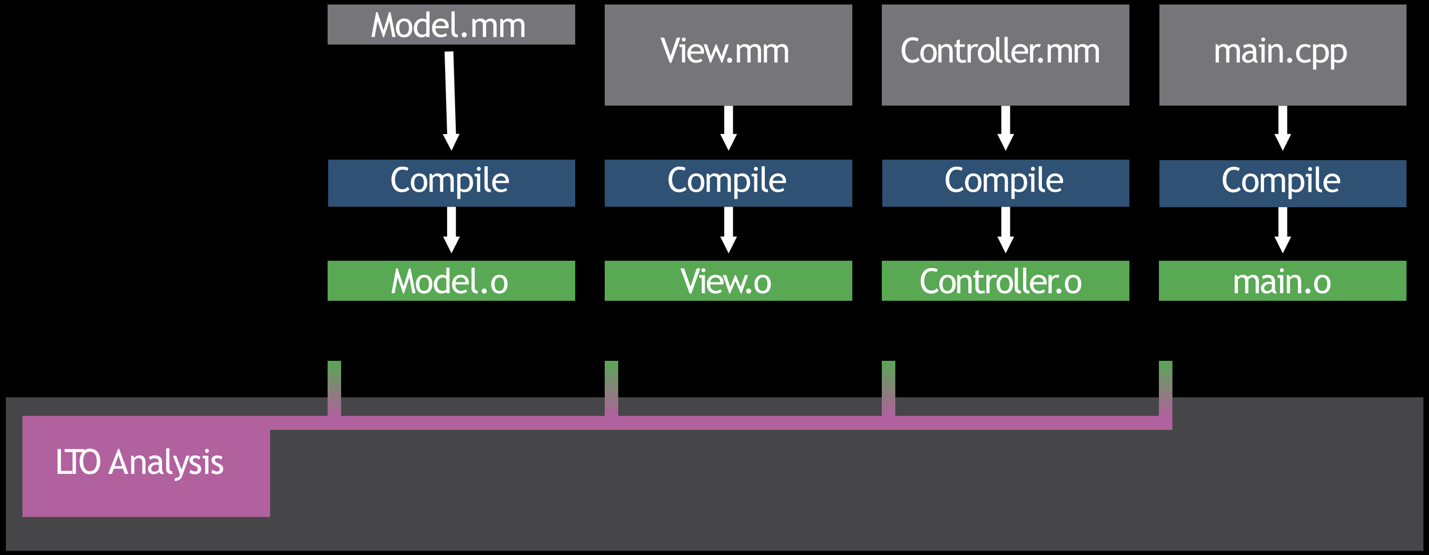
Incremental LTO Compilation Model



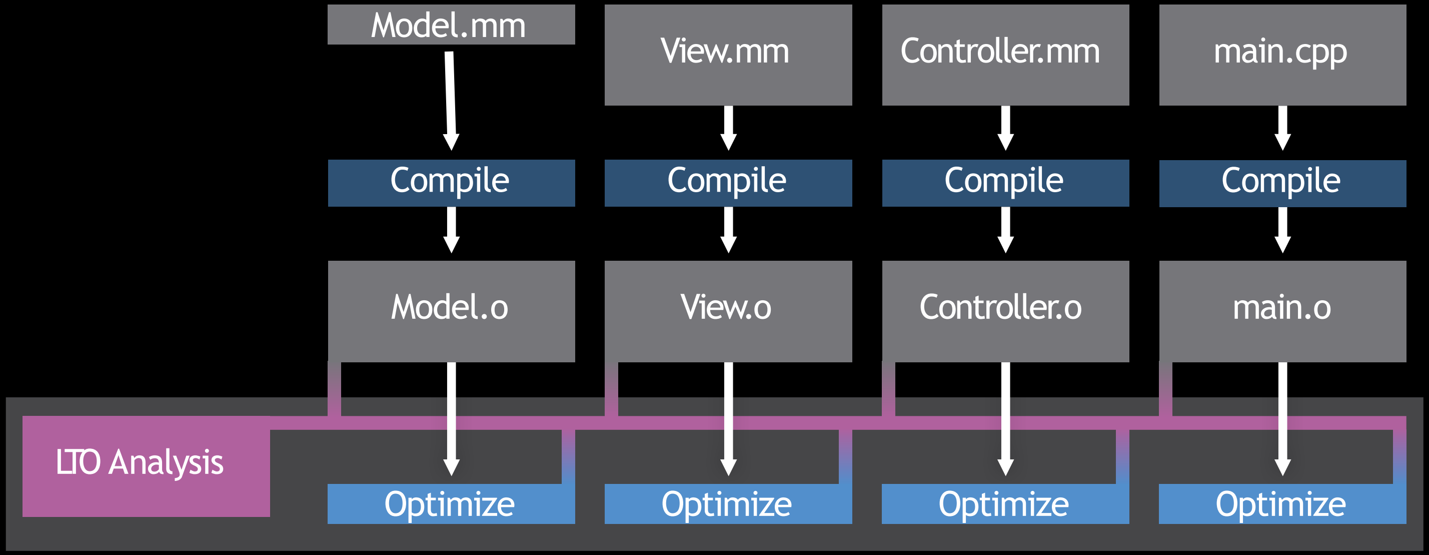
Incremental LTO Compilation Model



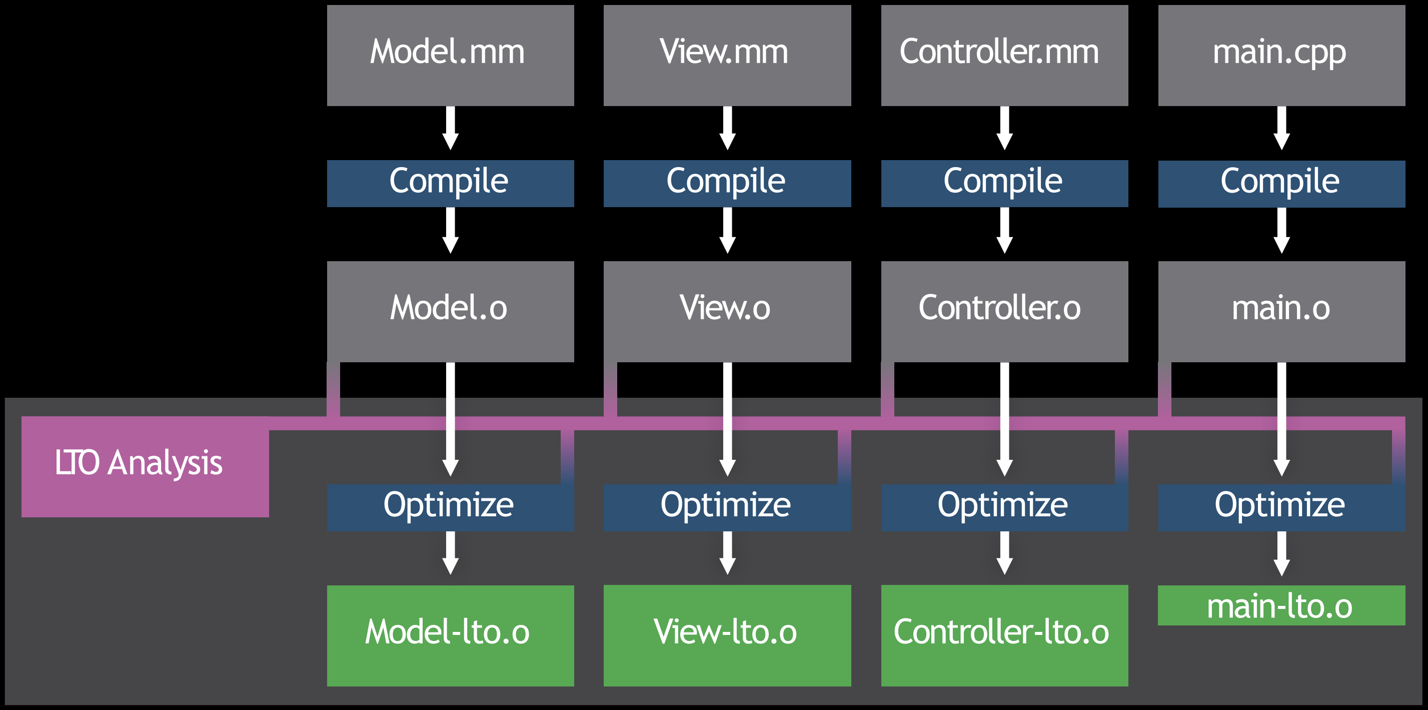
Incremental LTO Compilation Model


Model.mm
View.mm
Controller.mm
main.cpp
Compile
Compile
Compile
Compile
Model.o
View.o
Controller.o
main.o
LTO Analysis

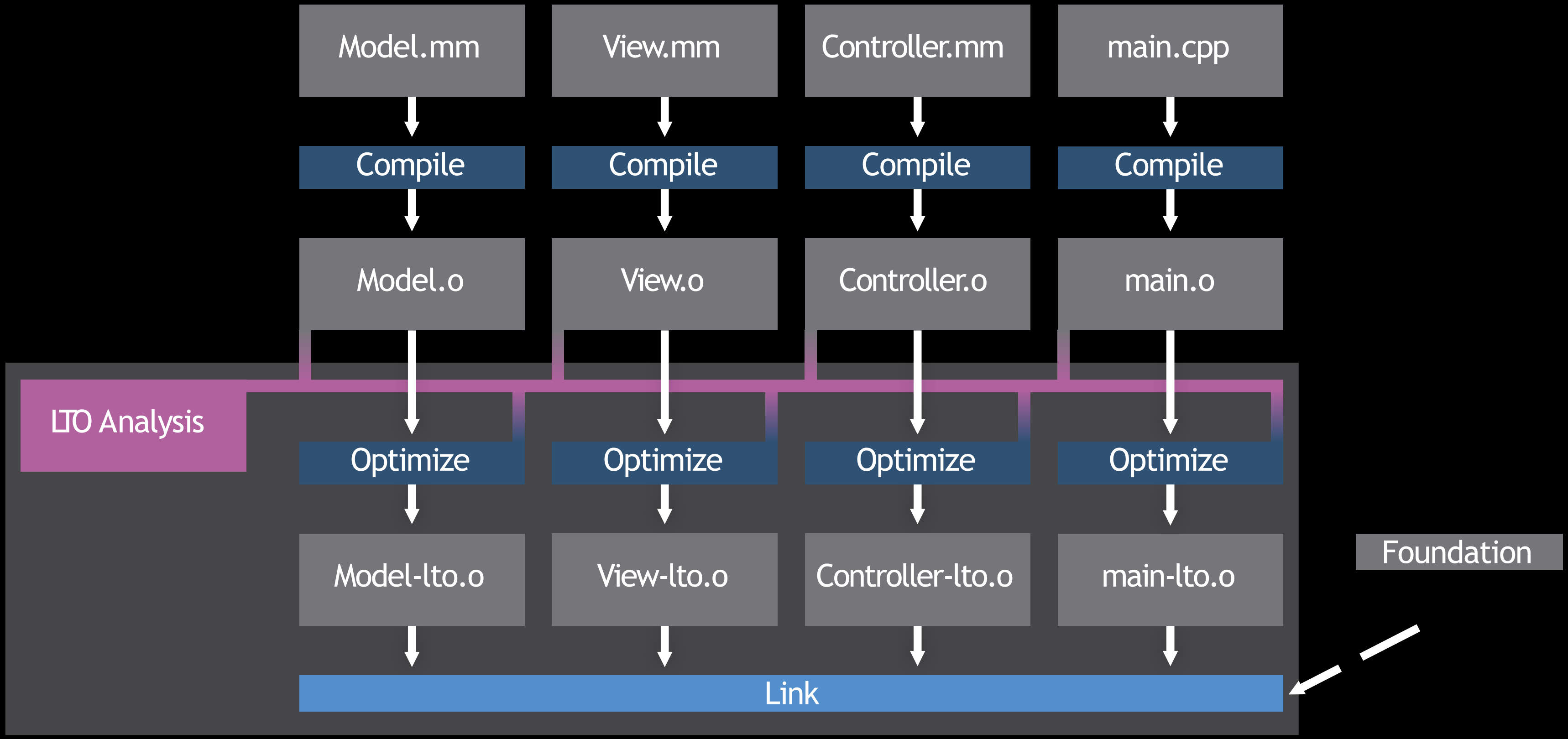
Incremental LTO Compilation Model



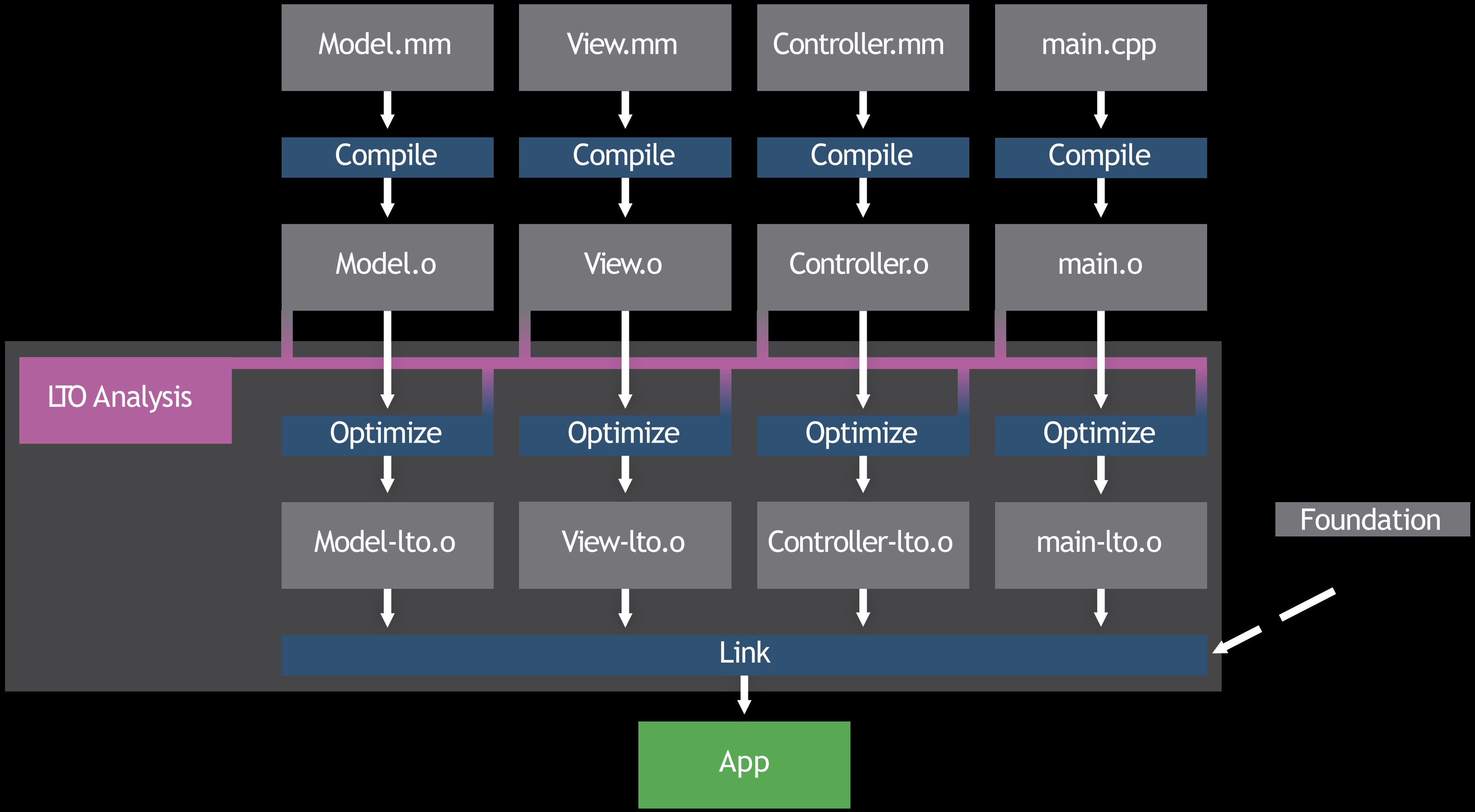
Incremental LTO Compilation Model



Incremental LTO Compilation Model

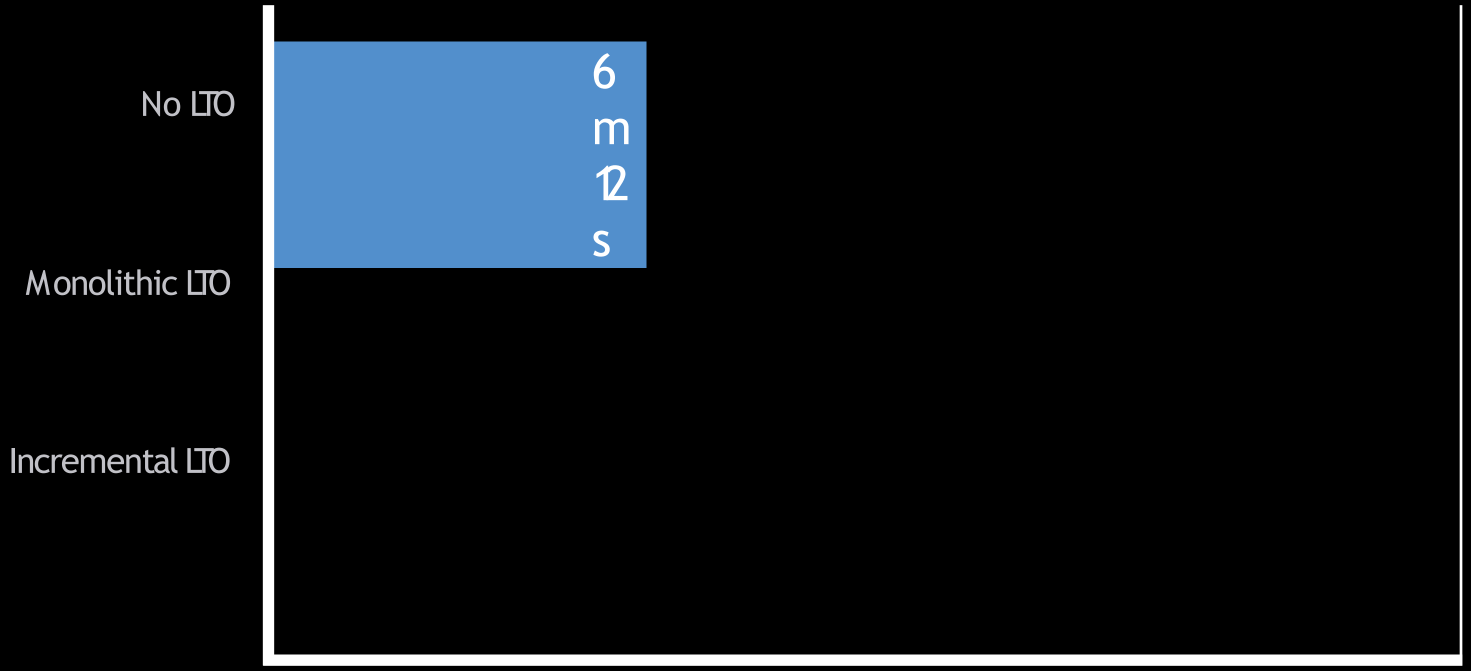


Incremental LTO Compilation Model



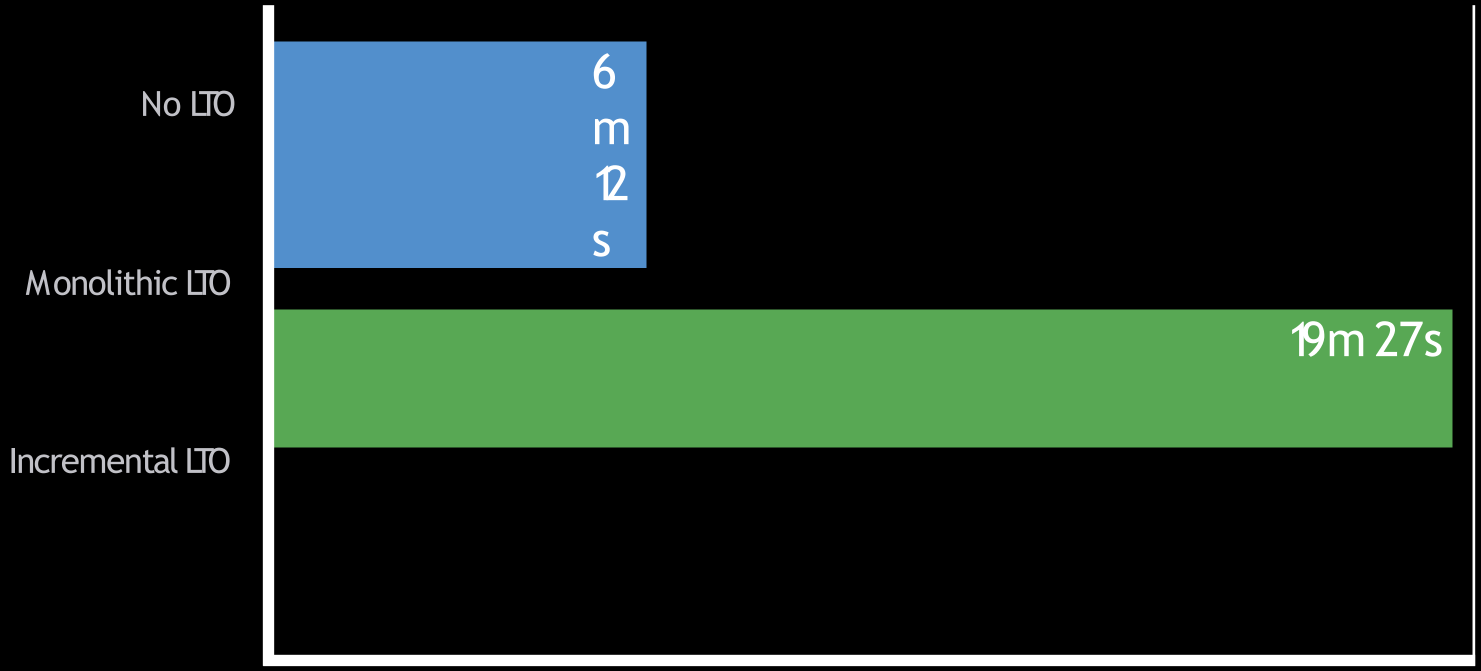
Time to Build a Large C++ Project

Smaller is better



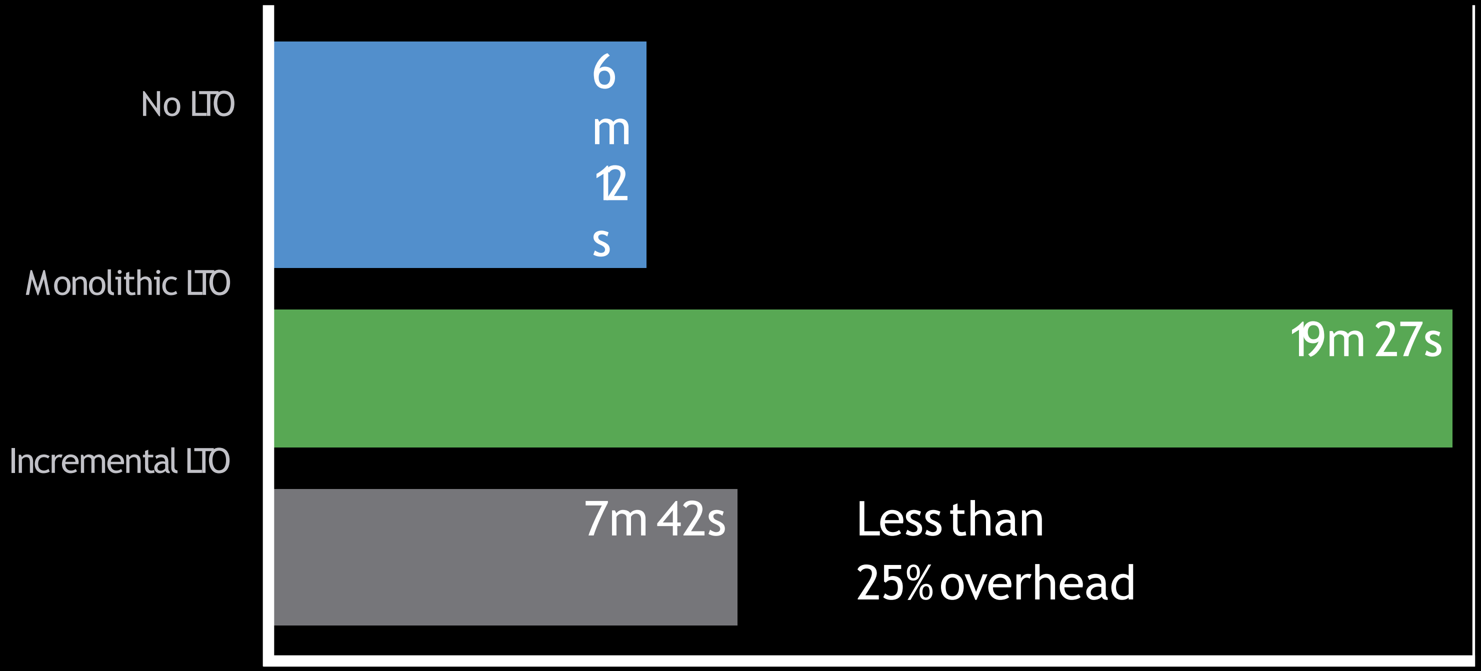
Time to Build a Large C++ Project

Smaller is better



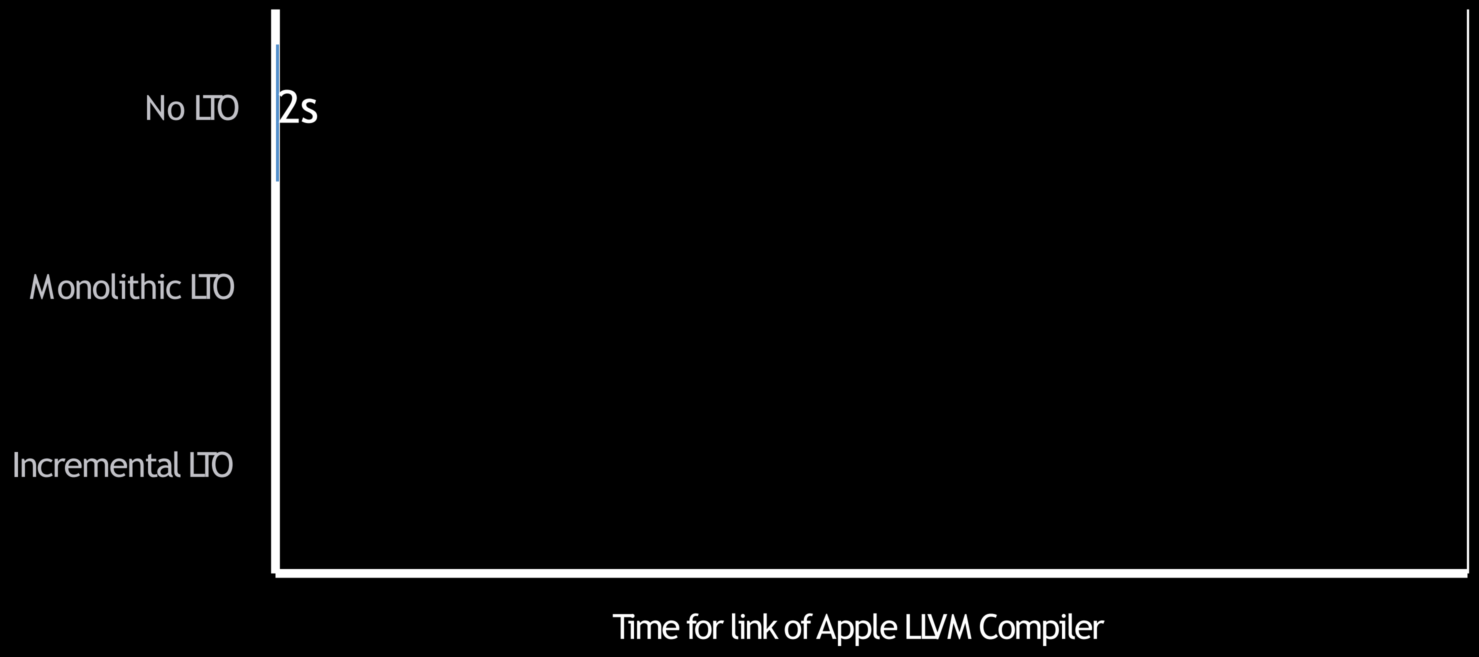
Time to Build a Large C++ Project

Smaller is better



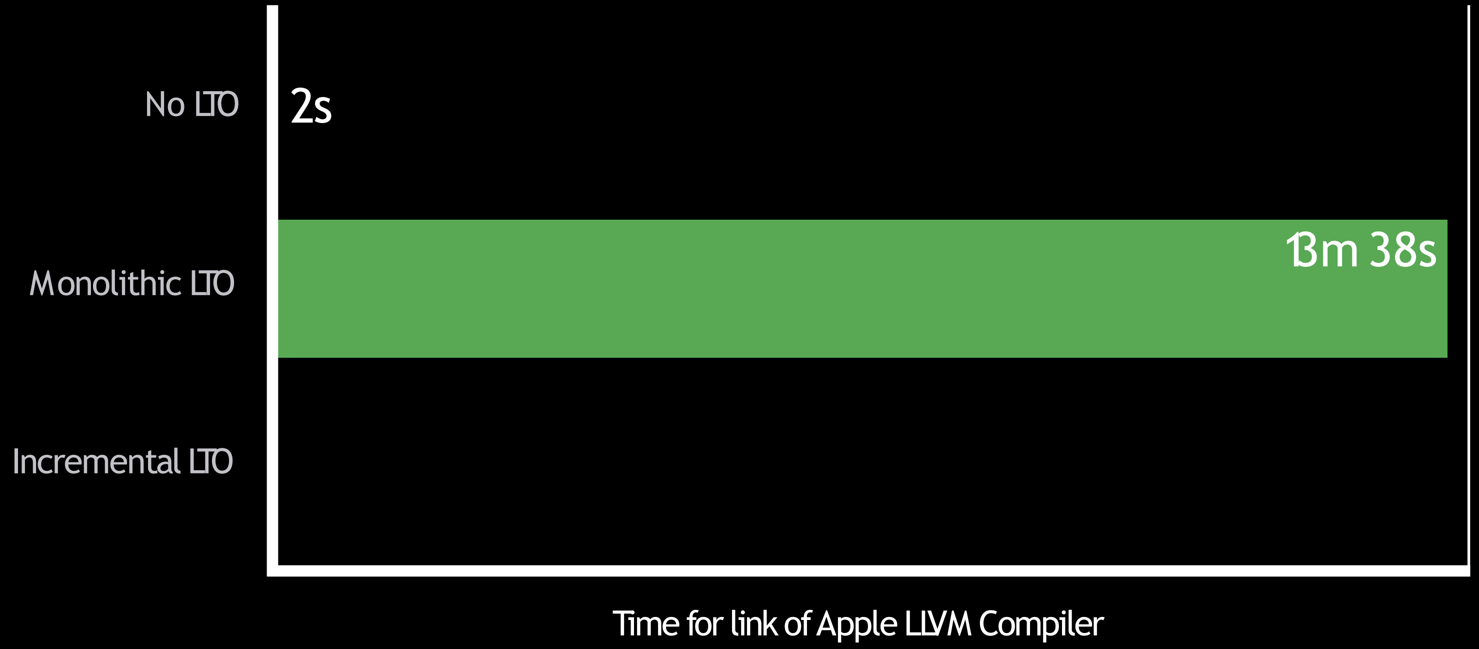
Time to Link a Large C++ Project

Smaller is better



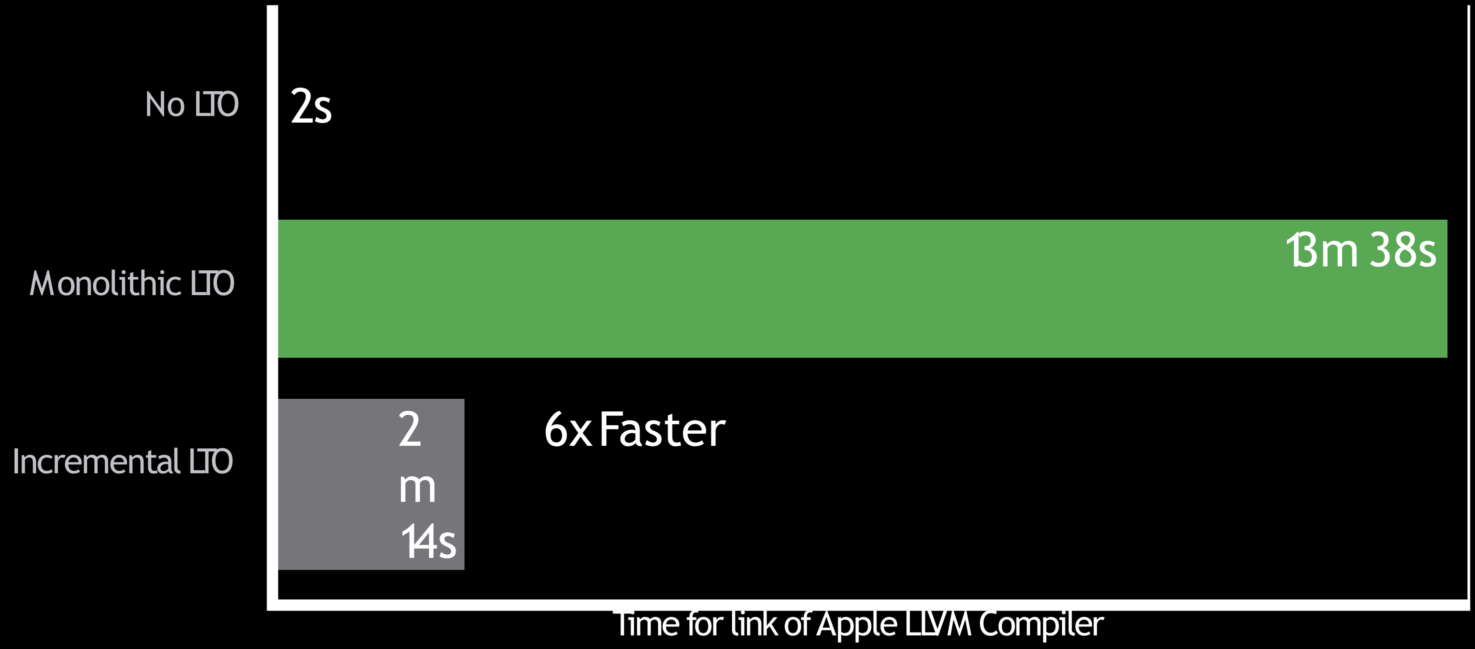
Time to Link a Large C++ Project

Smaller is better



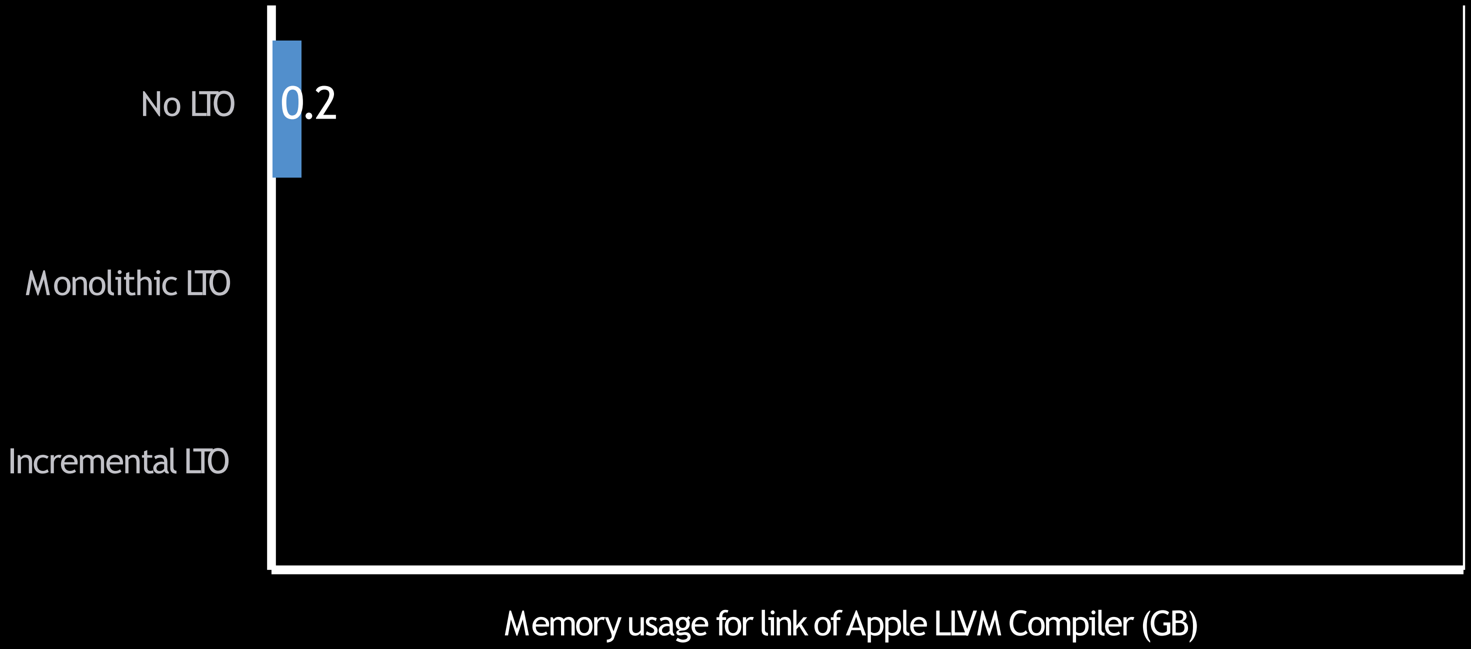
Time to Link a Large C++ Project

Smaller is better



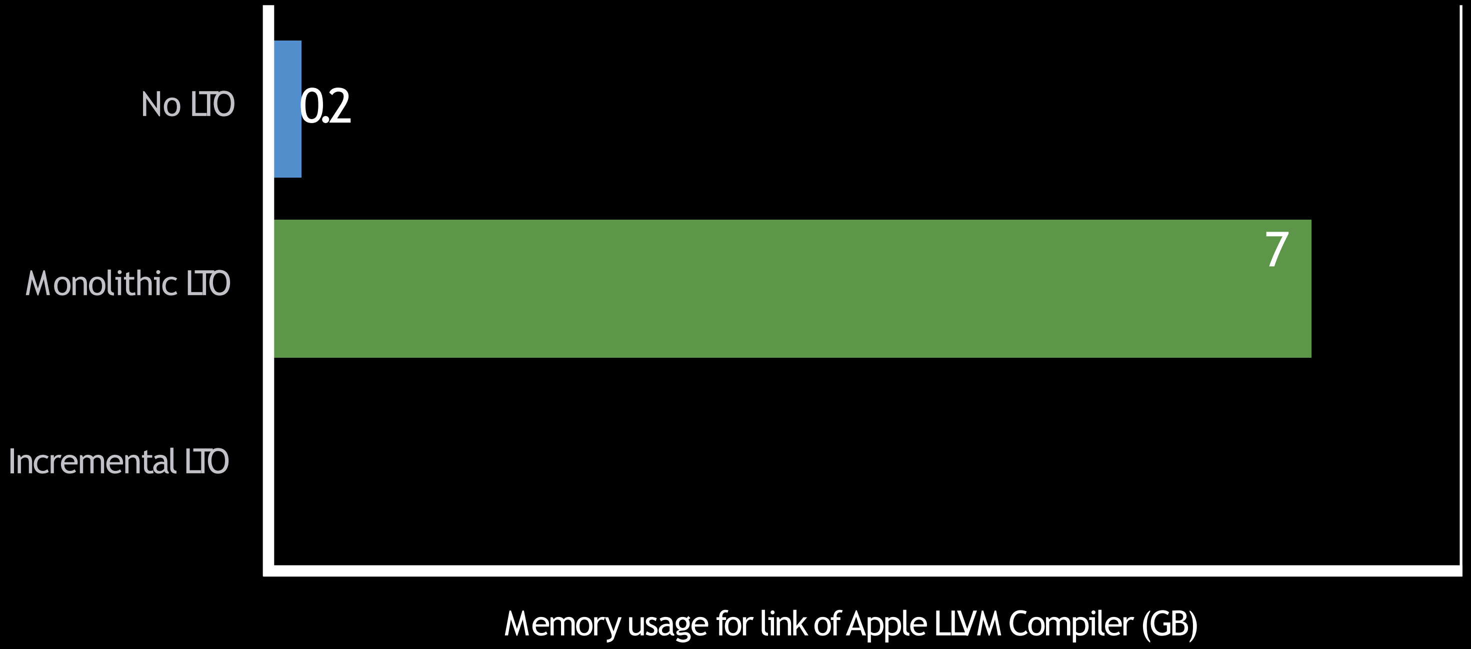
Memory to Link a Large C++ Project

Smaller is better



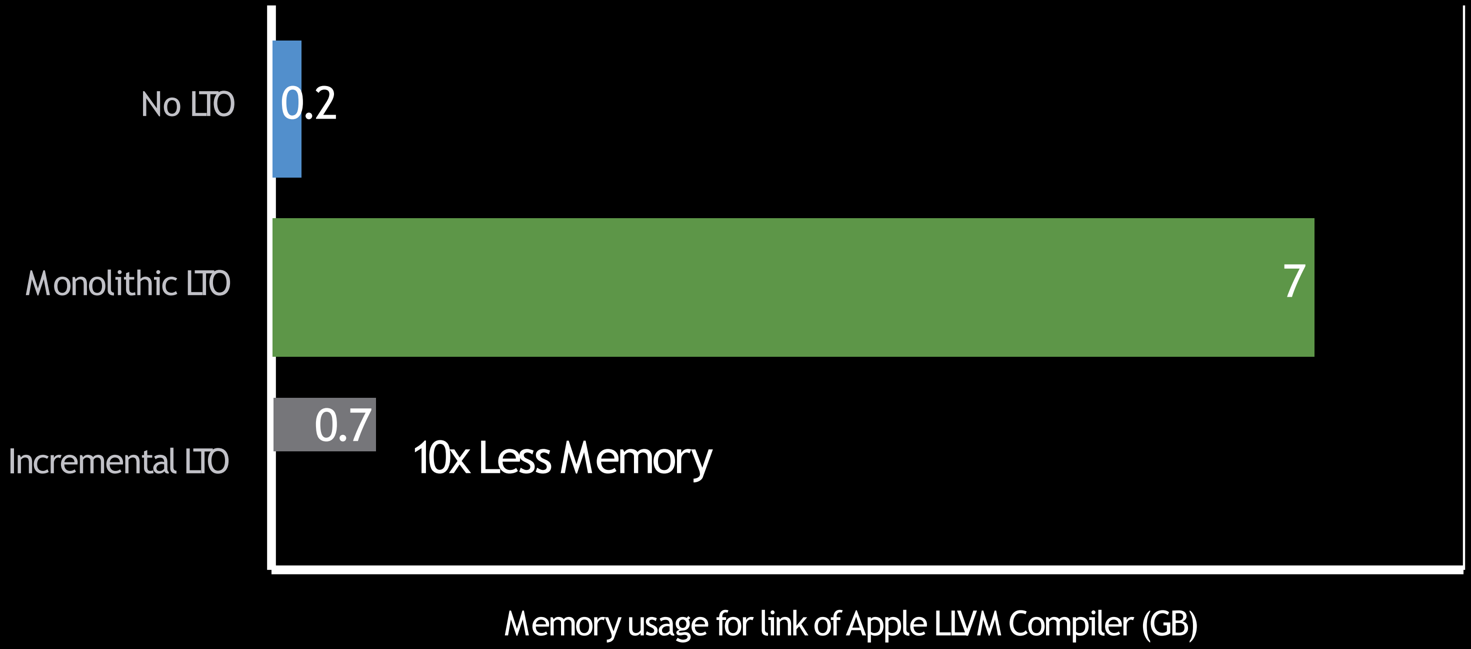
Memory to Link a Large C++ Project

Smaller is better

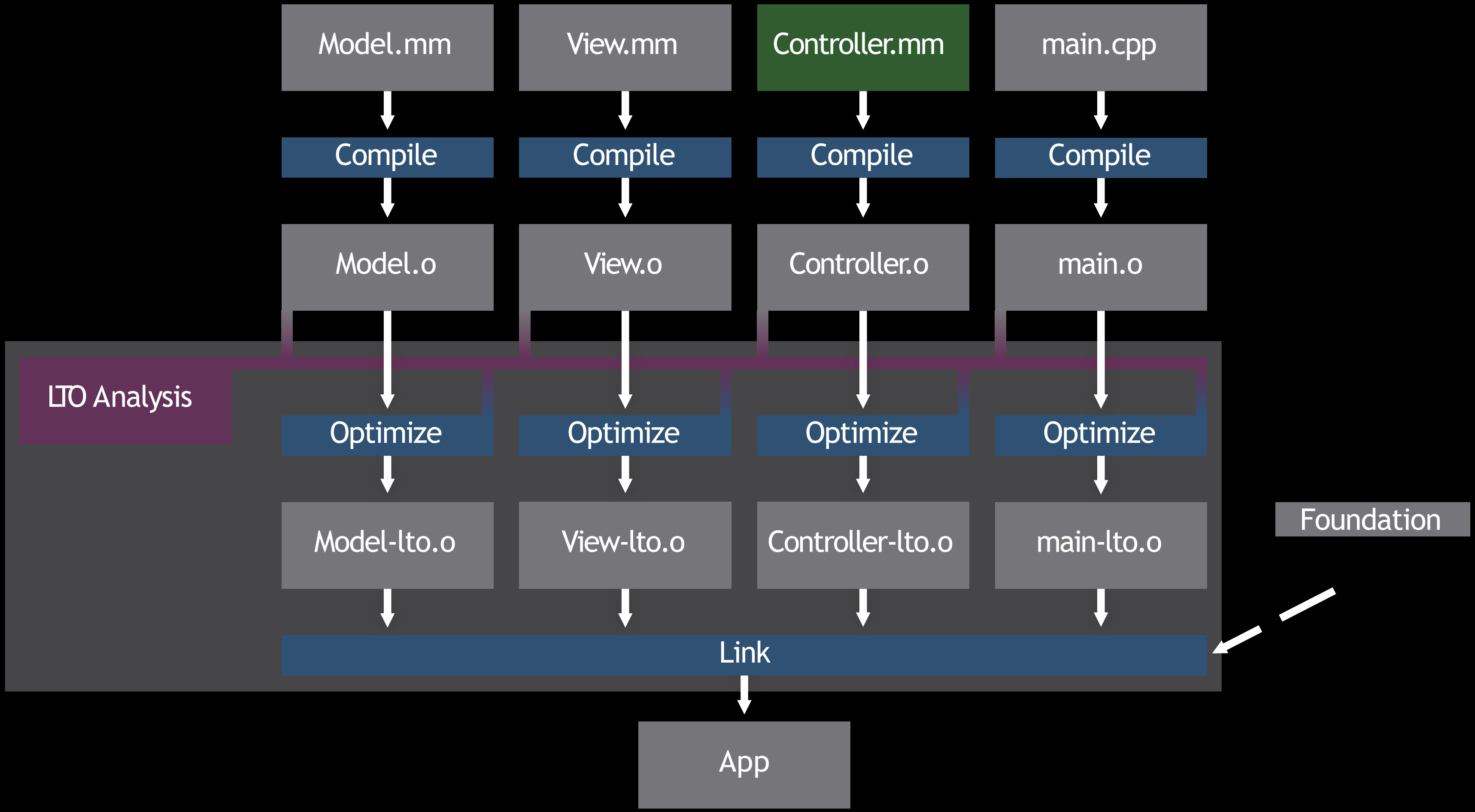


Memory to Link a Large C++ Project

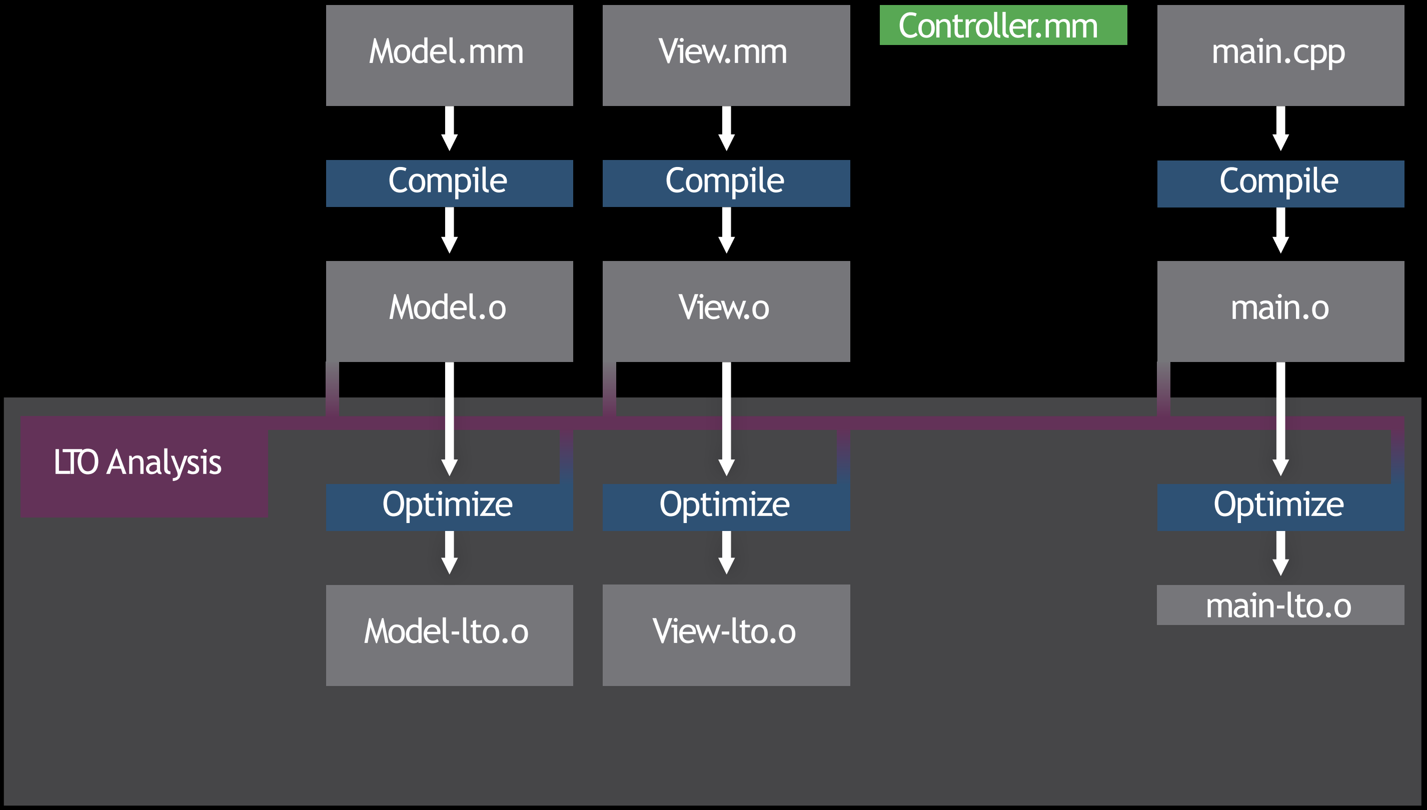
Smaller is better



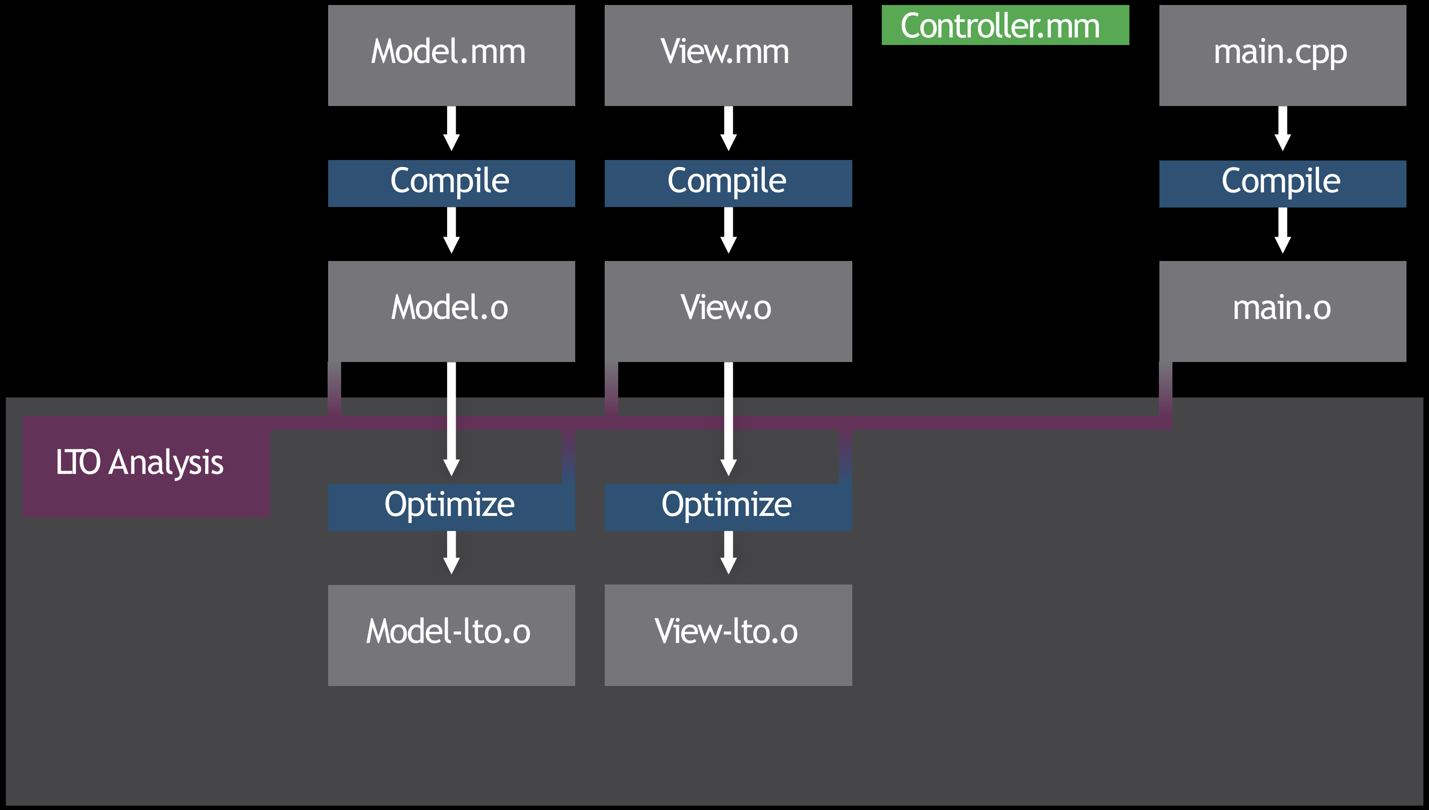
Example of Incremental Build



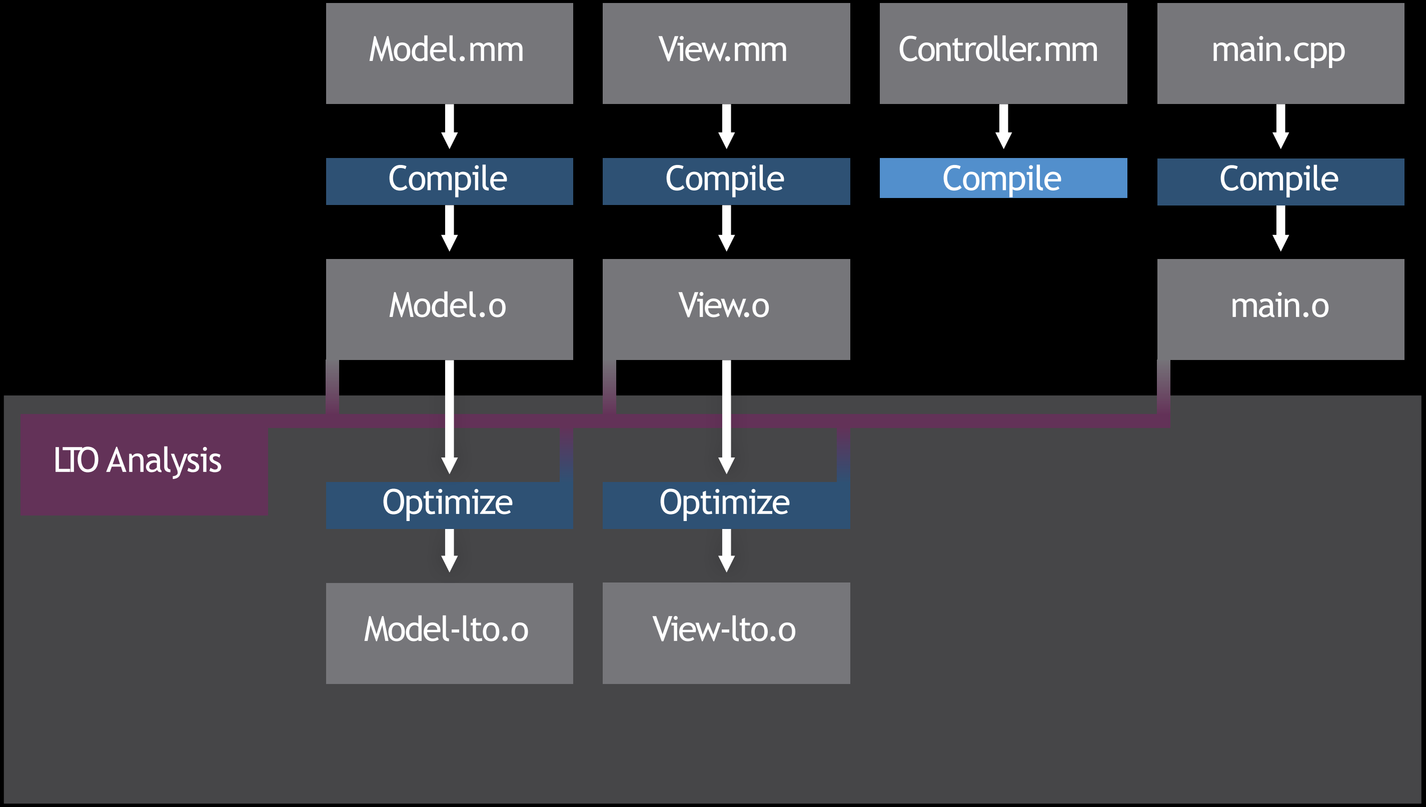
Example of Incremental Build



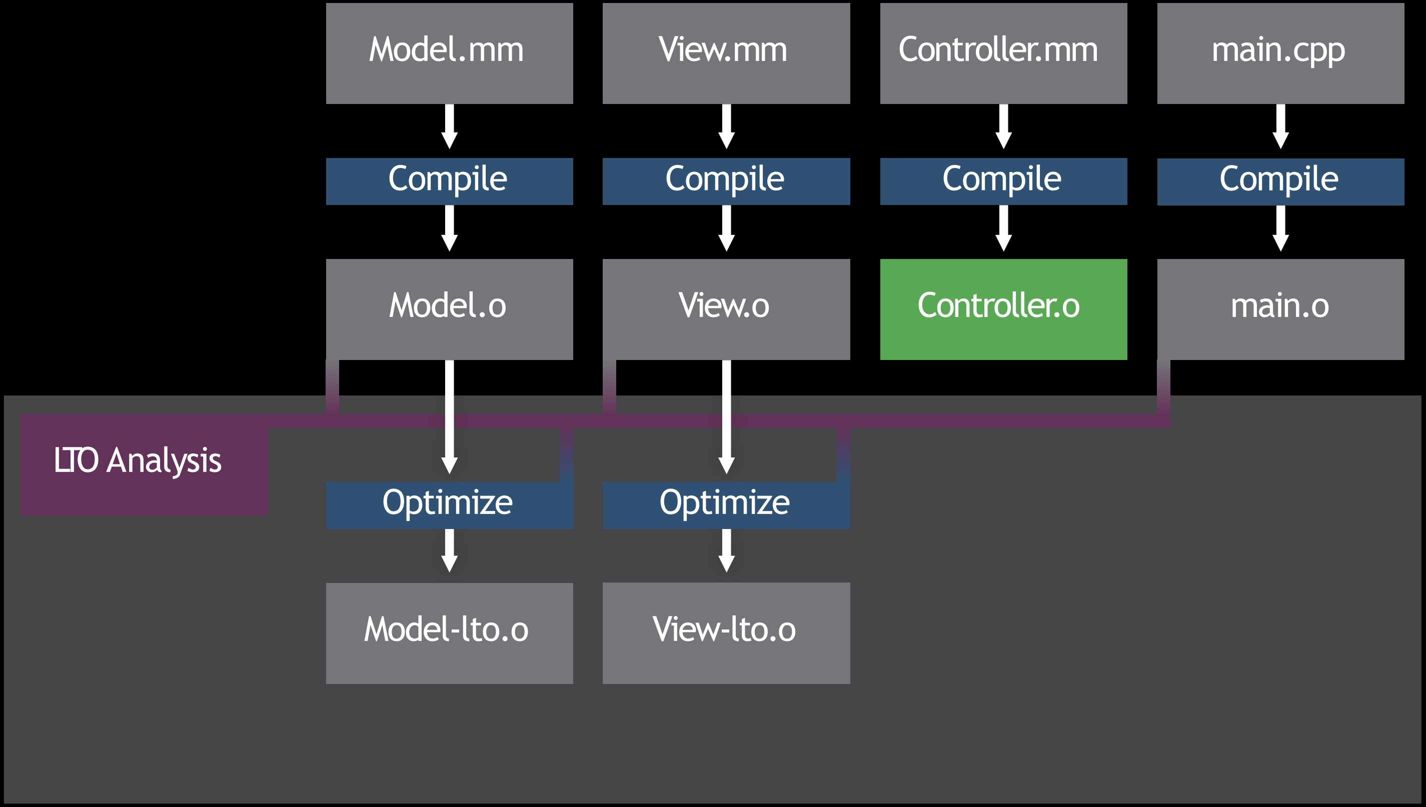
Example of Incremental Build



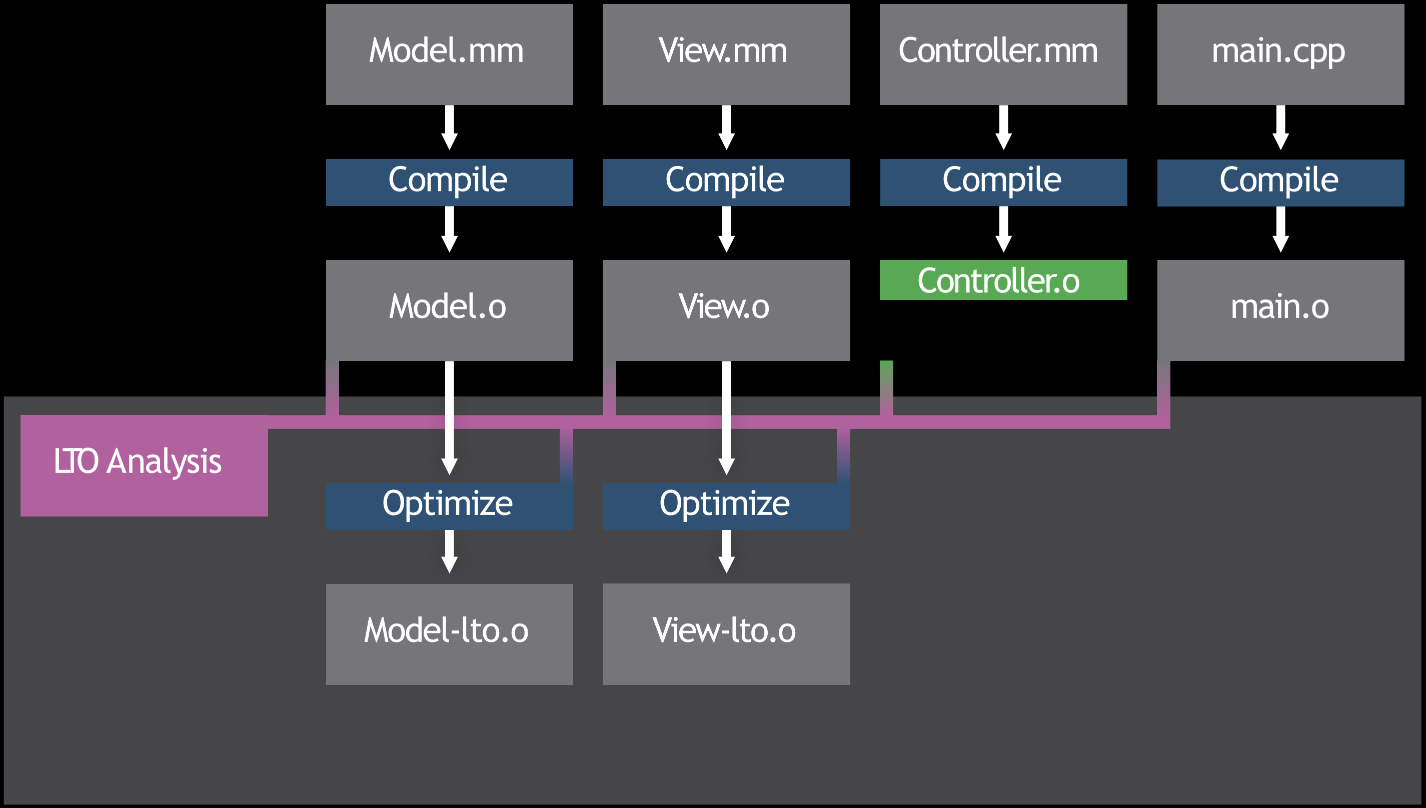
Example of Incremental Build



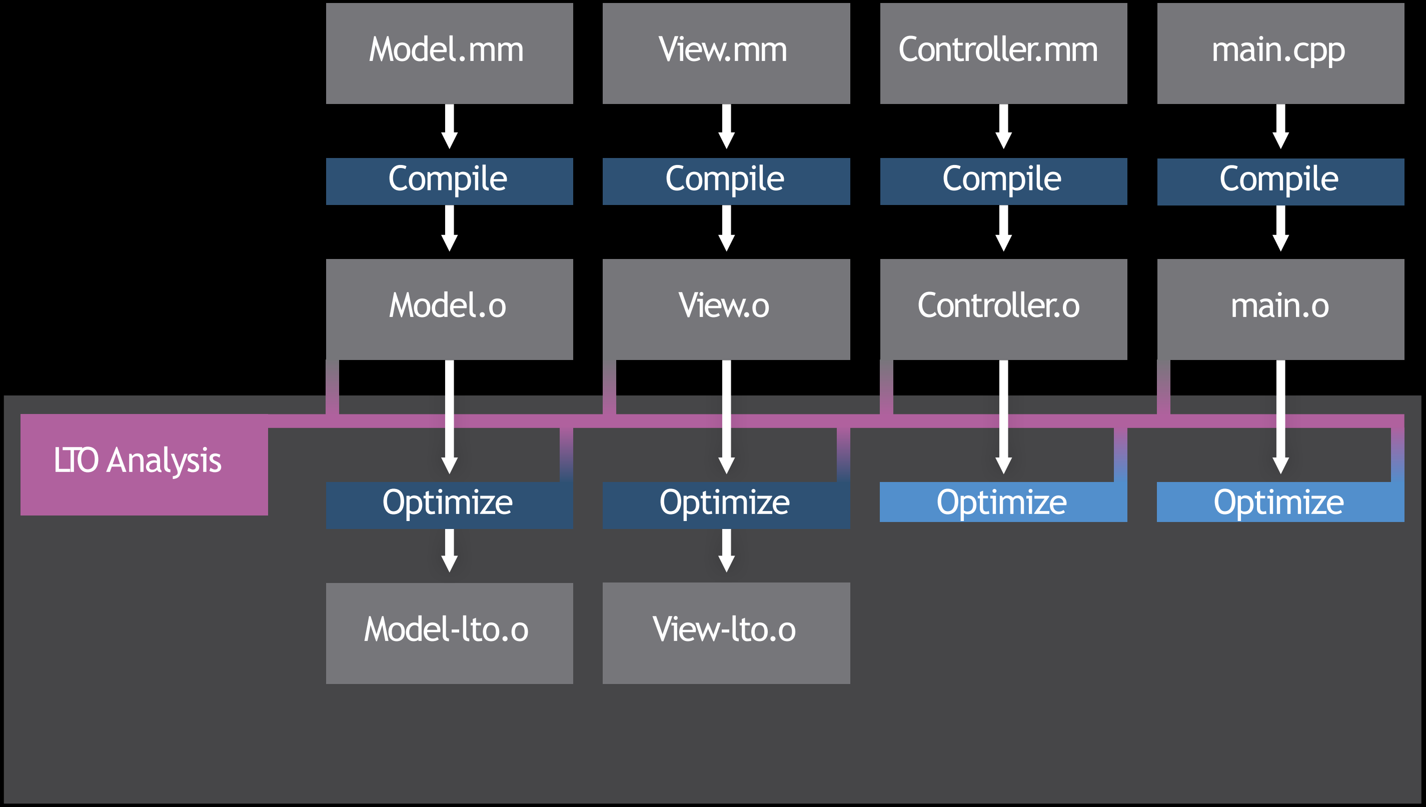
Example of Incremental Build



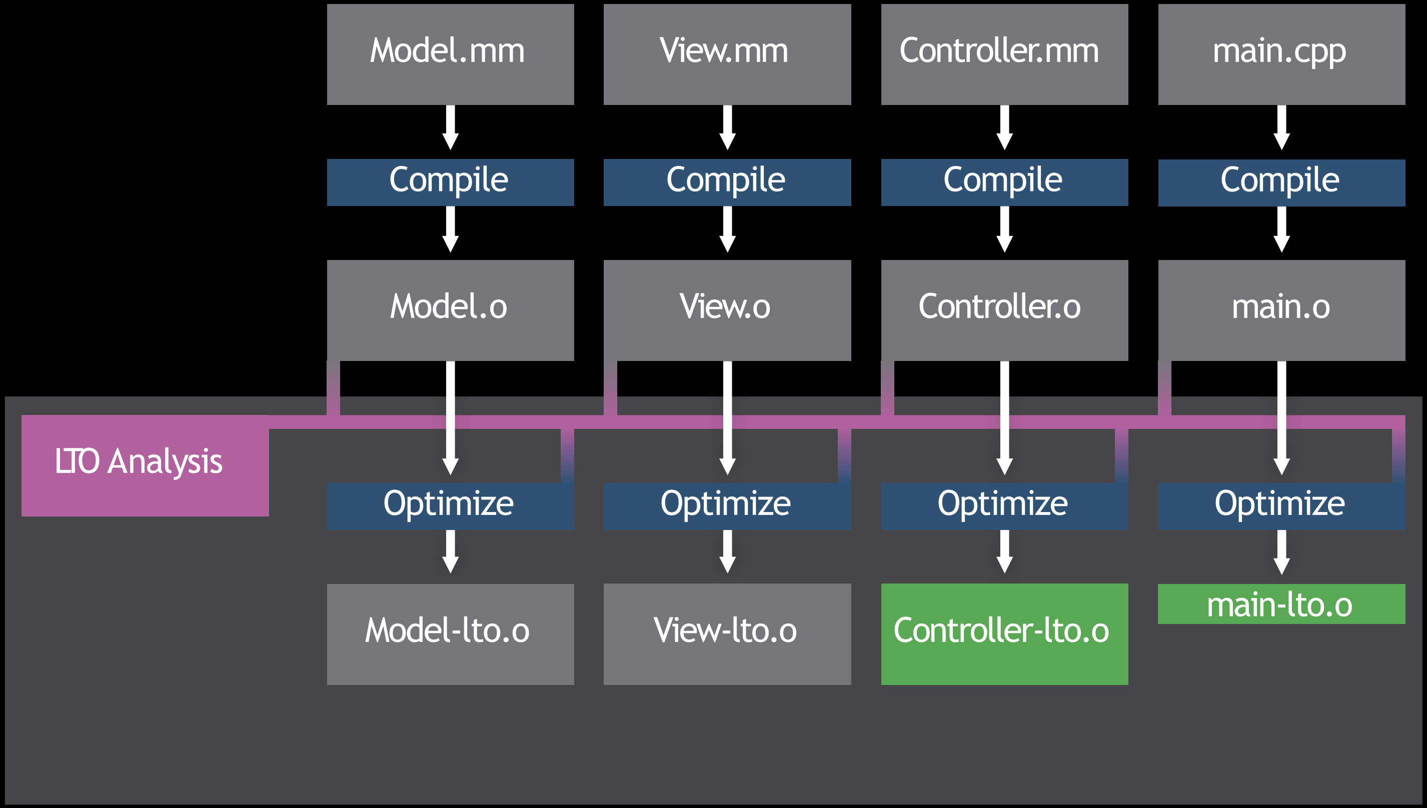
Example of Incremental Build



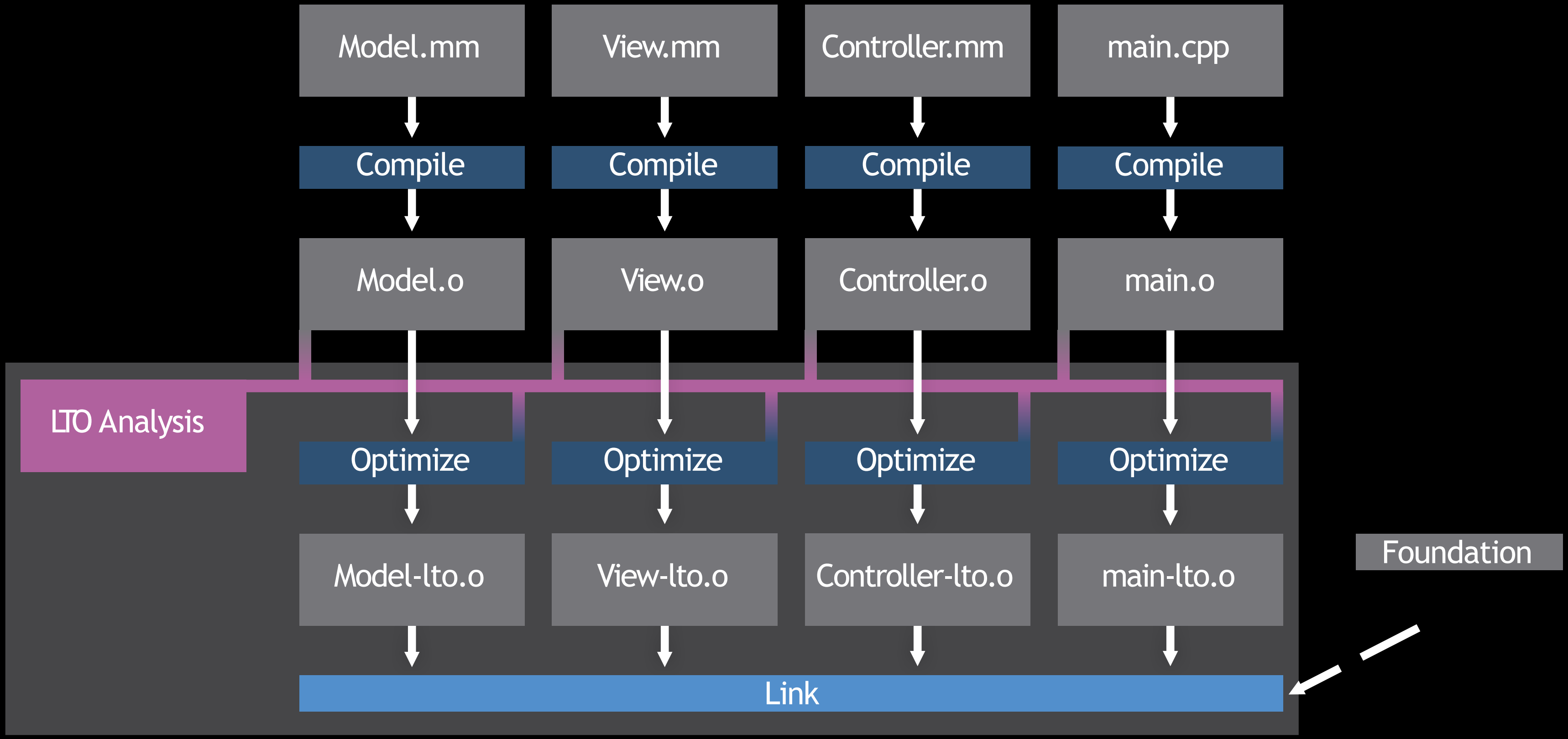
Example of Incremental Build



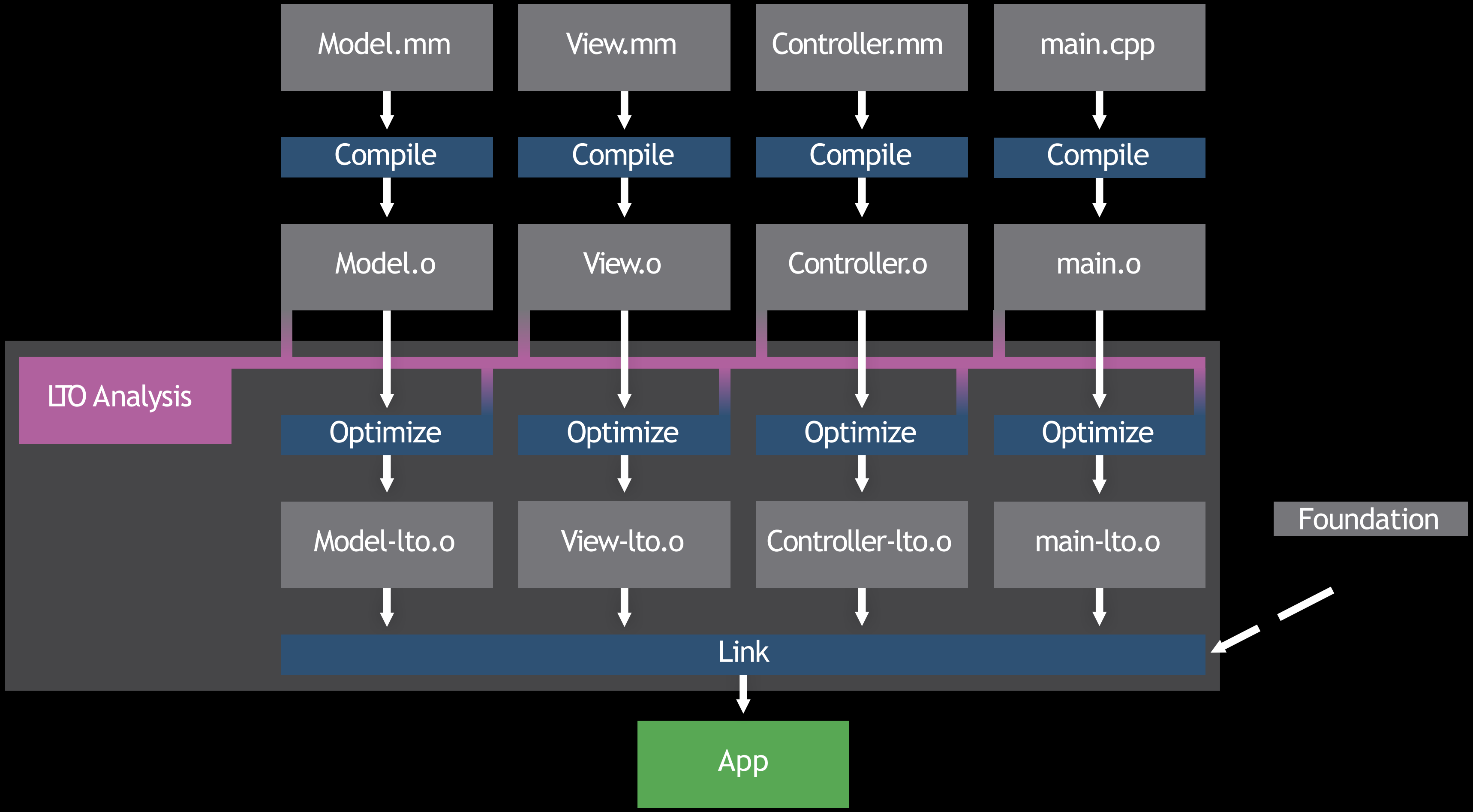
Example of Incremental Build



Example of Incremental Build

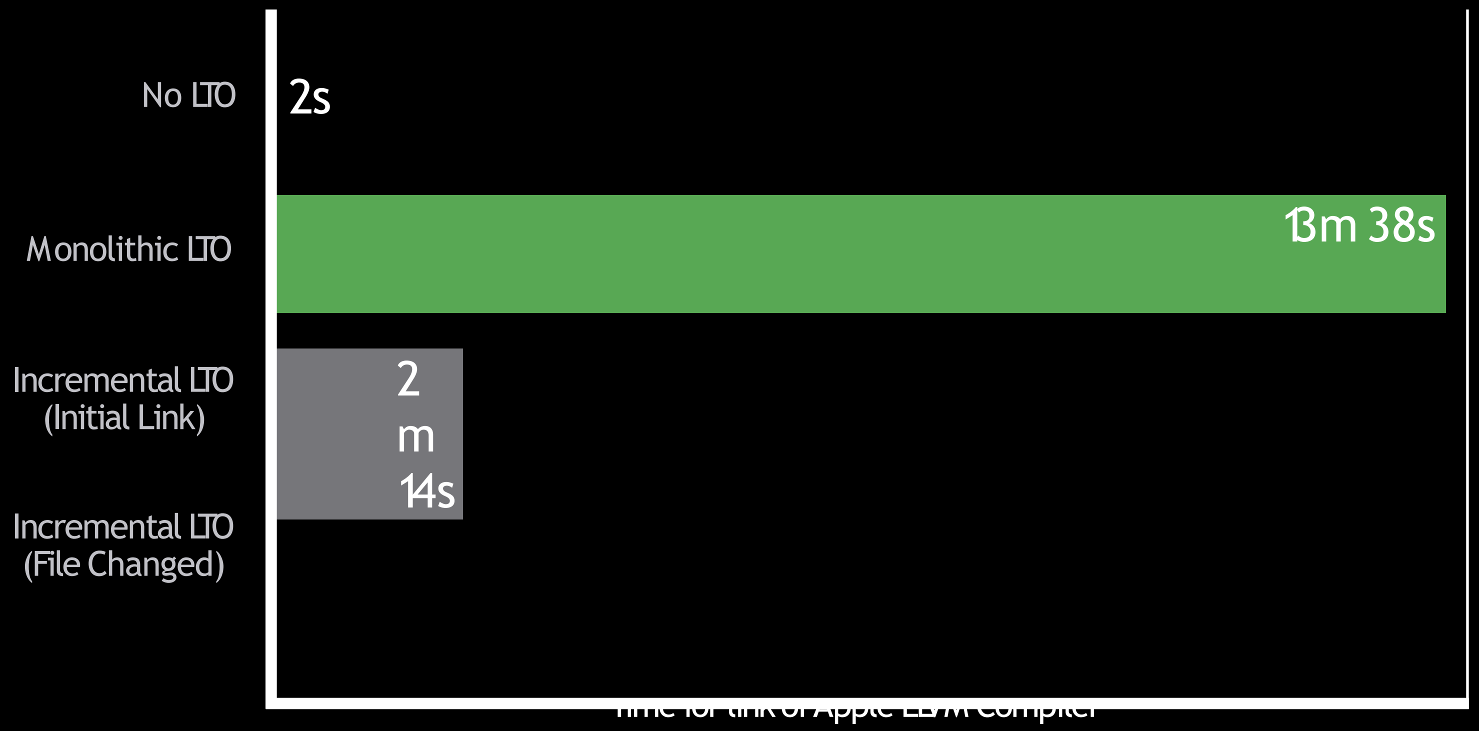


Example of Incremental Build



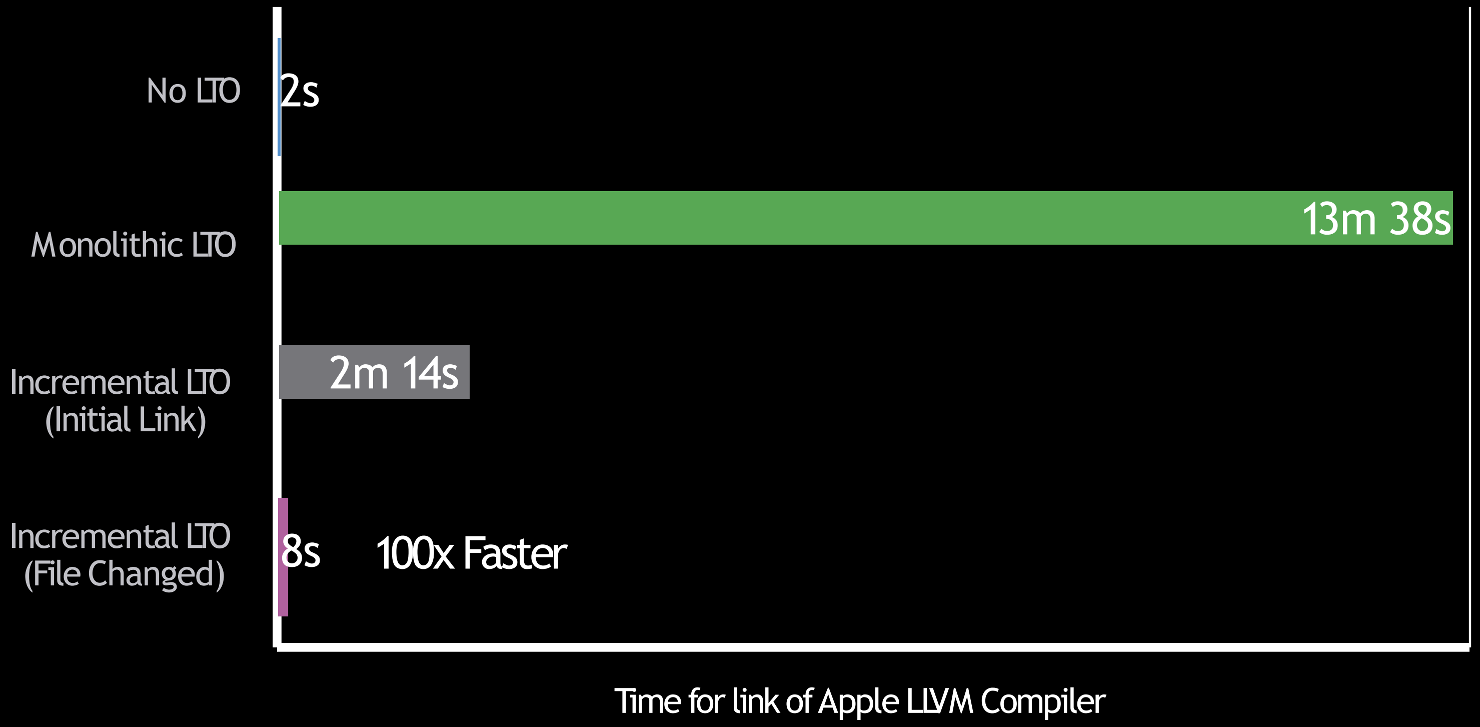
Incremental Link of Large C++ Project

Smaller is better



Incremental Link of Large C++ Project

Smaller is better

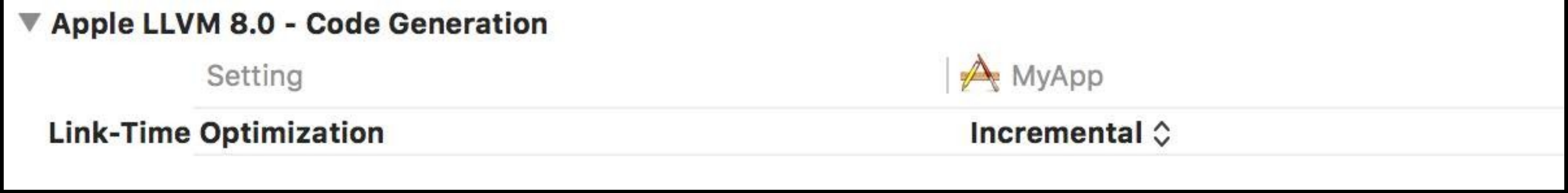


Enable Incremental LTO

Runtime performance similar to Monolithic LTO

Memory usage 10x smaller than Monolithic LTO

Incremental link almost as fast as No LTO



▼ Apple LLVM 8.0 - Code Generation

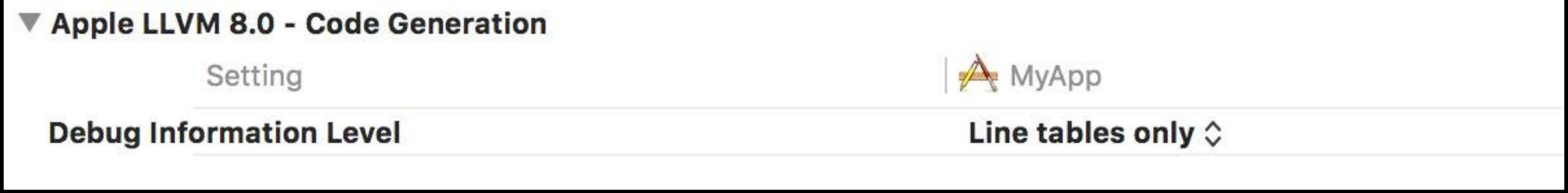
Setting	MyApp
Link-Time Optimization	Incremental ◇

LTO and Debug Info

Recommendation

Use `-gline-tables-only` with large C++ projects

- Shorter compile time
- Smaller memory footprint
- Same rich backtraces at runtime



▼ Apple LLVM 8.0 - Code Generation

Setting	MyApp
Debug Information Level	Line tables only ◊