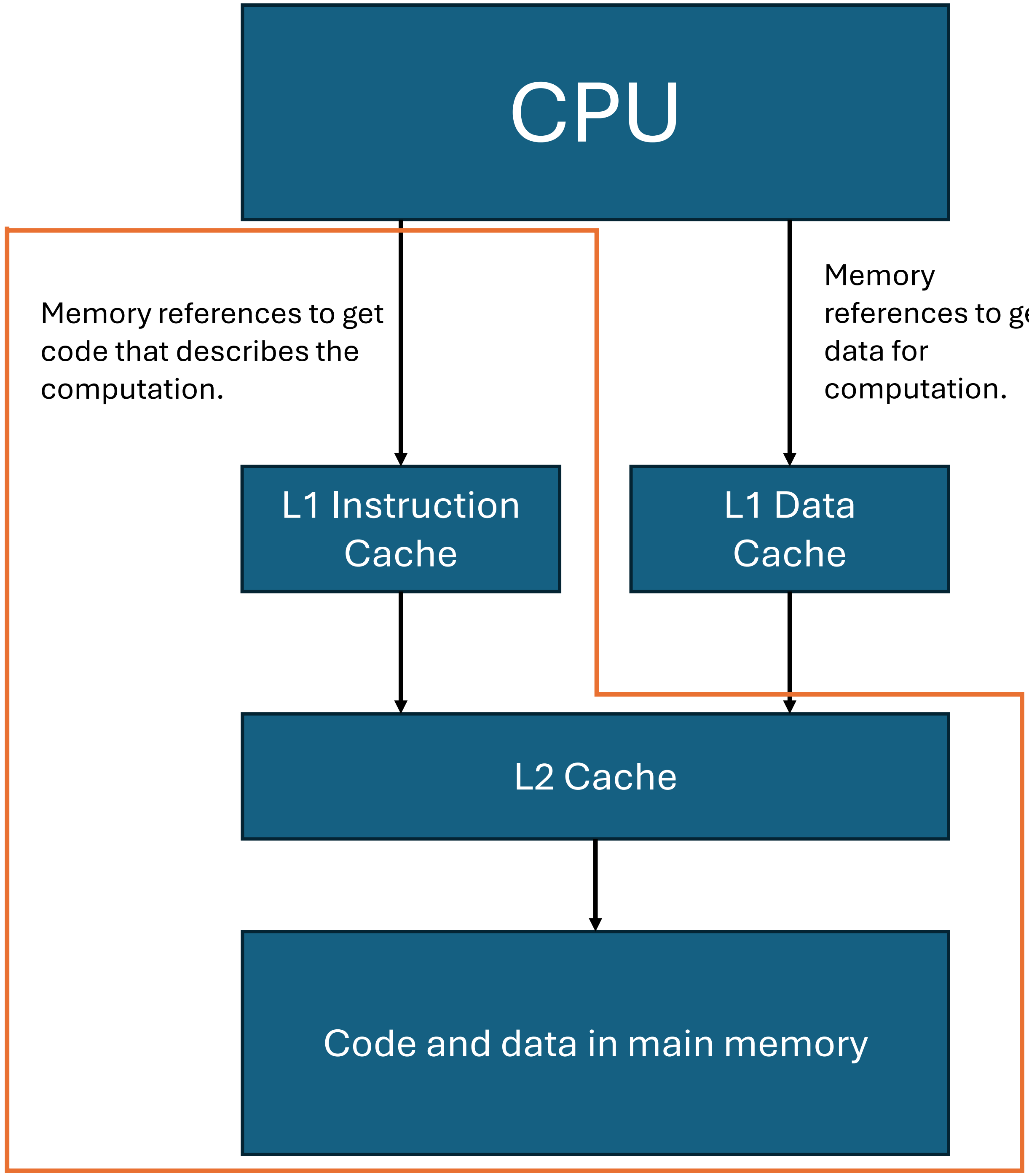
Bringing Profile Guided Function Placement to AIX

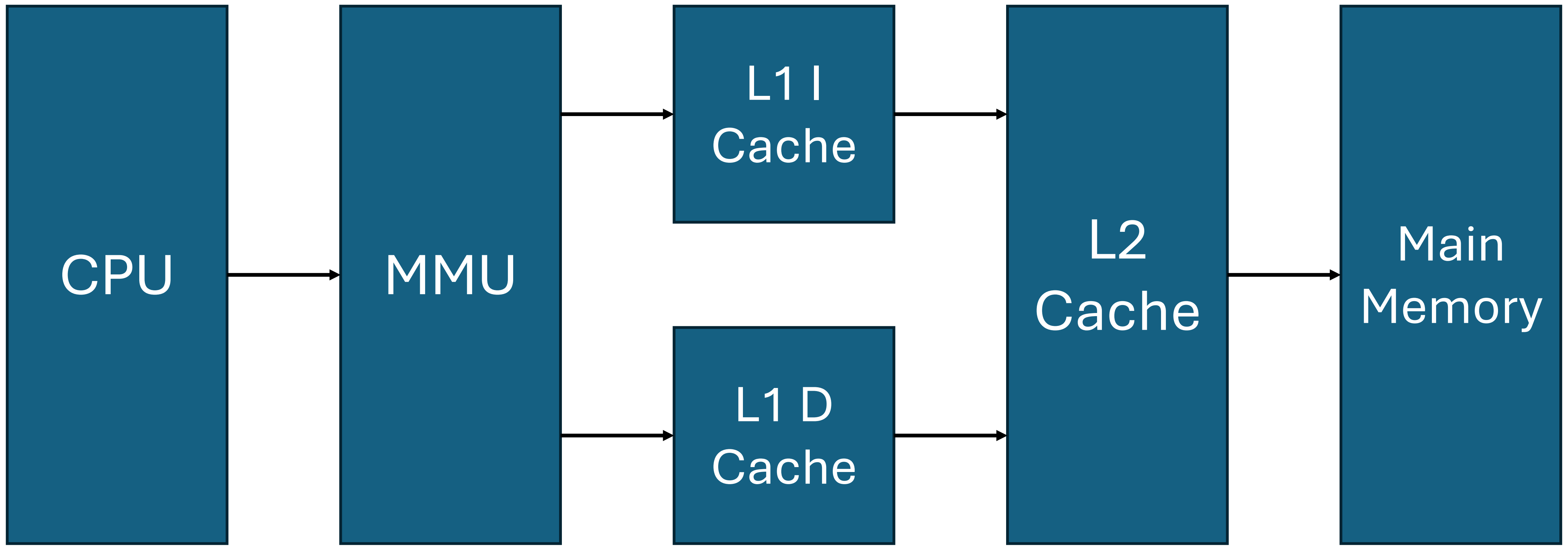
Dhanrajbir Singh Hira

J. Nelson Amaral

University of Alberta


CPU
Memory references to get code that describes the computation.
Memory references to get data for computation.
L1 Instruction Cache
L1 Data Cache
L2 Cache
Code and data in main memory

- There is another crucial part of our model that is missing, and that is the mapping between virtual and memory addresses.
- The addresses that our program accesses are virtual memory address and the MMU needs to map those addresses to actual physical address in the main memory to access them.



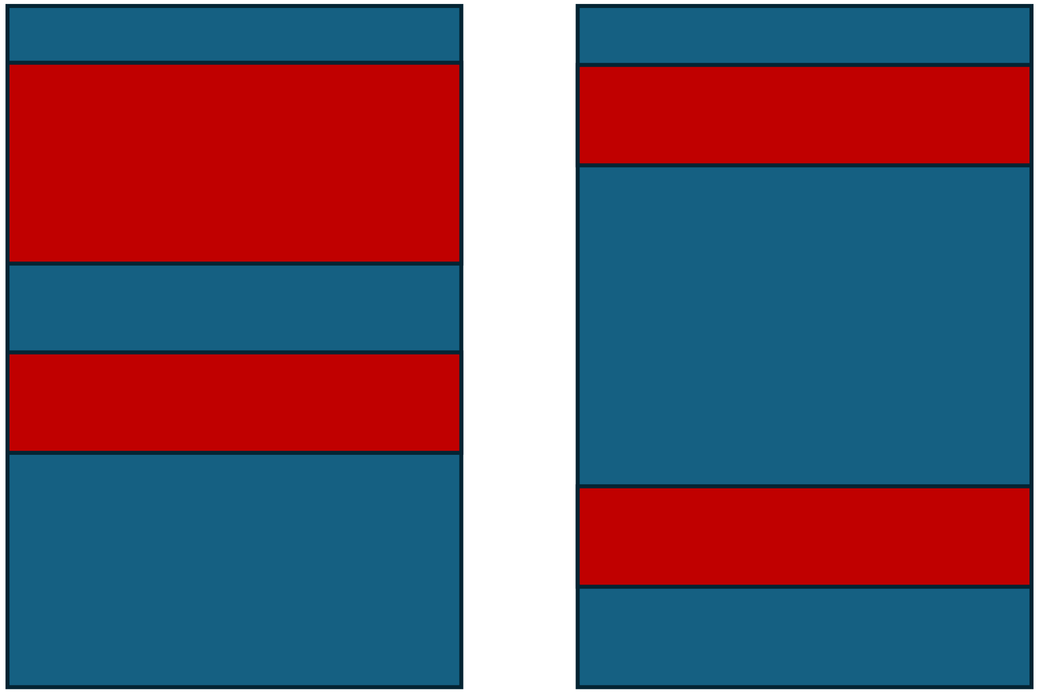
Inside the MMU

- Virtual addresses are translated to physical addresses at page level granularity.
- There are multiple different organization on how this mapping (page table) is stored.
  - Radix tables
  - Hash based
- Needs to be performed on every memory access, so needs to be fast. How do CPU designers make things fast? Caching.
- The structure that caches these translations is called TLB or translation lookaside buffer.

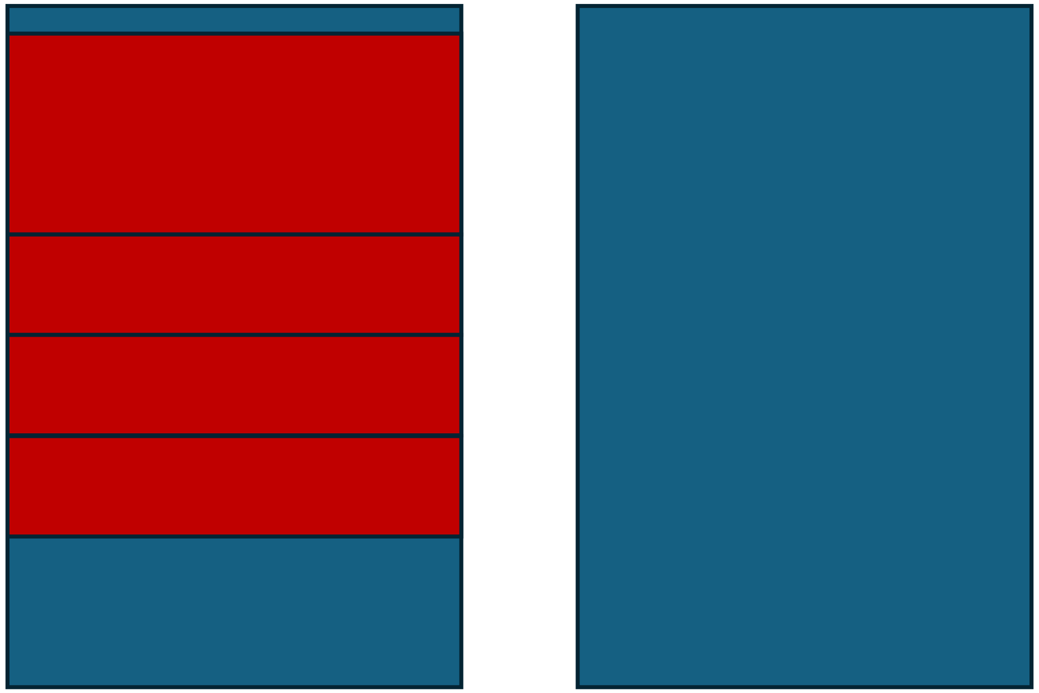
- TLB entries are a scarce resource especially in large-scale multiprocess applications.
- TLB misses together with I-cache misses means that often in large scale application it’s a challenge to keep the CPU fed with instructions.
- In most programs there are certain sections of code that are executed more frequently (called hot) than others (called cold)
- We can reduce the effective working set of the program code by arranging these hot section in a way that minimizes the number of memory pages they span,
- This effectively reduces the number of entries the TLB must cache simultaneously to keep the CPU fed with instructions.

Example

- Let’s assume our program spans 2 memory pages.



Unoptimized organization



Optimized organization

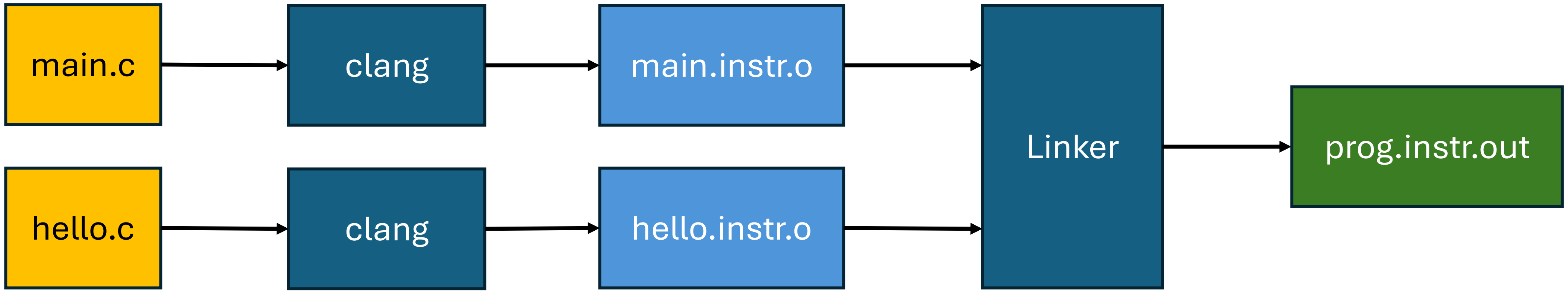
RISC ISAs have another problem.

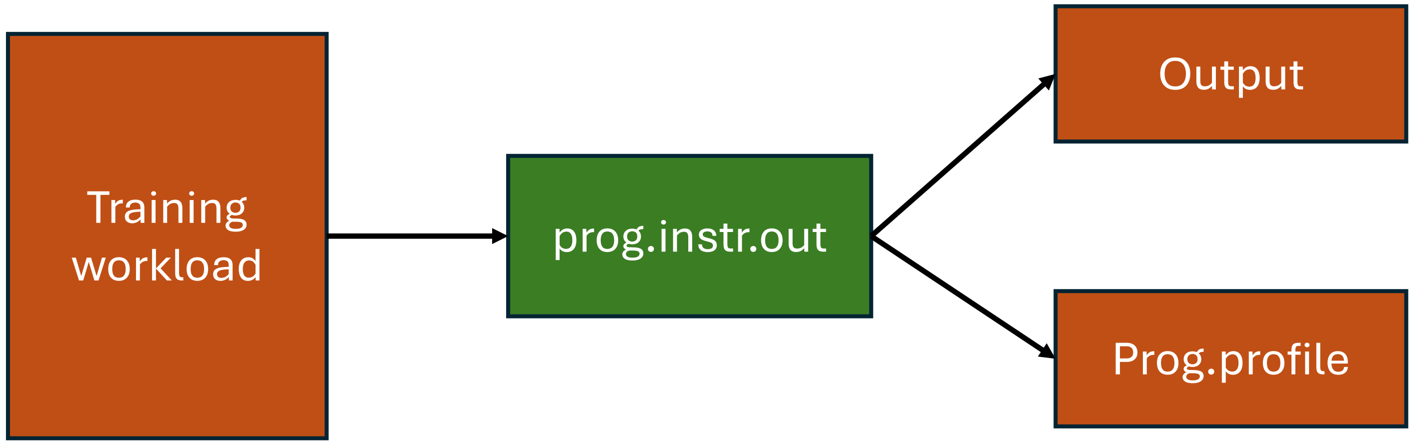
- Since each instruction is fixed length, there are only so many bits left to store the offset of the branch target.
- This means the instruction sequence for calling your function depends on how far away in your memory space is the callee from the caller.
- If there are a lot of trampolines on your hot path, they can take up a non-trivial proportion of execution time.

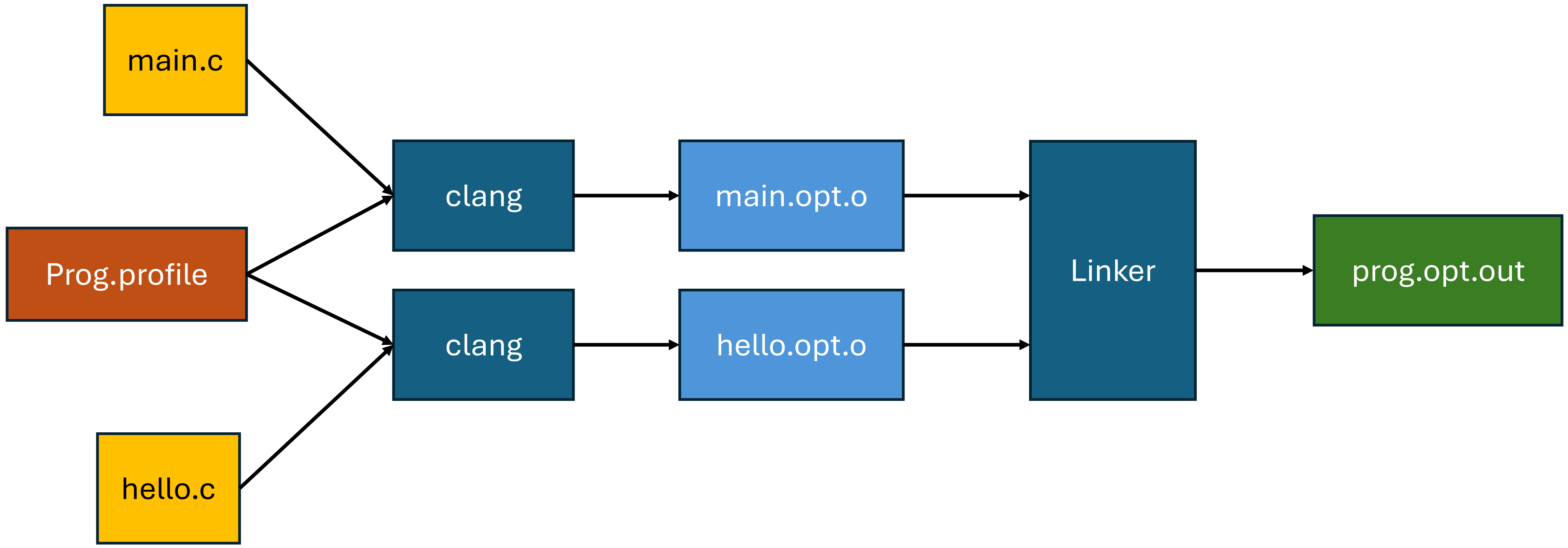
Existing alternative solution

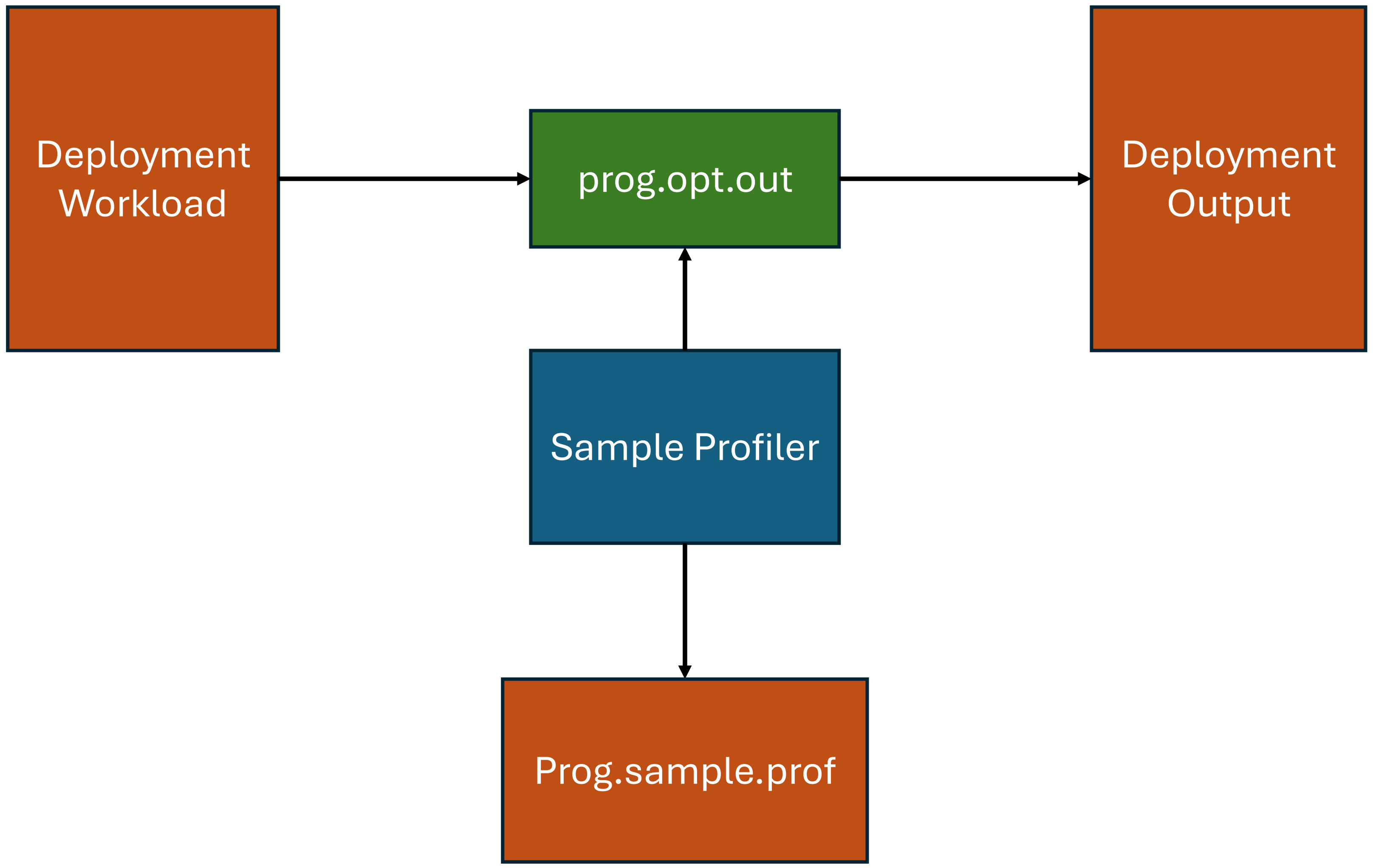
- llvm-BOLT
- BOLT is a binary optimizer that takes in a sample-based profile of the application and then optimizes a program binary based on that profile.
- Performs function ordering but also a bunch of other optimizations.
- Currently only supports x86 and AArch64 platforms.
- Let’s see an example workflow of optimizing a binary with BOLT.

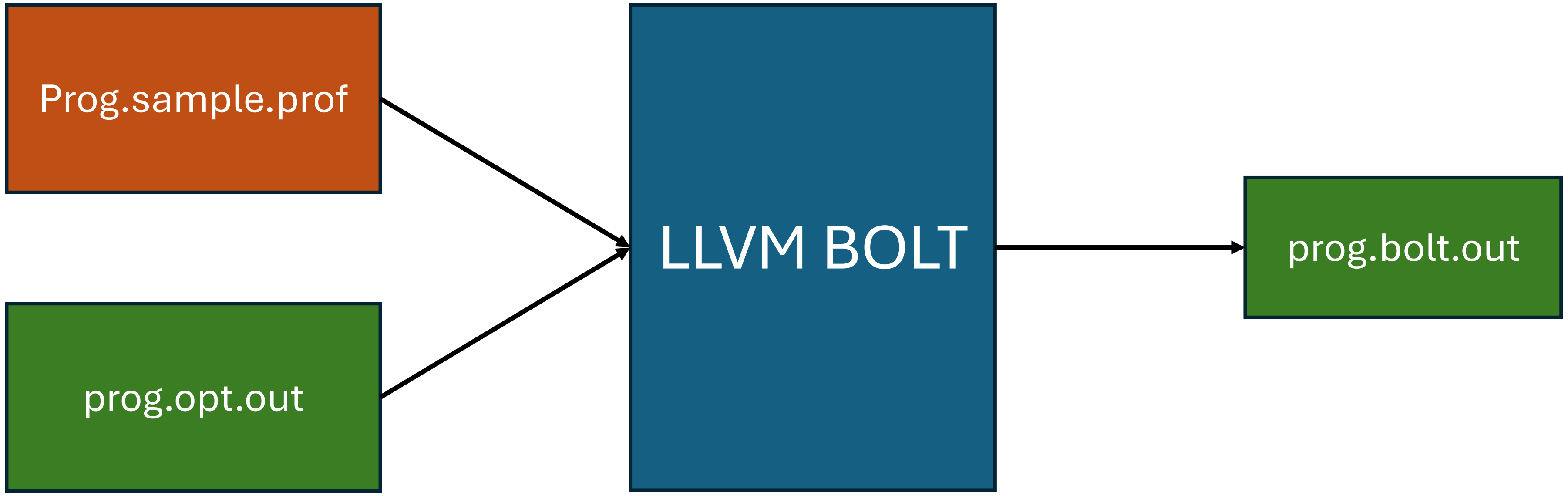
Example BOLT workflow






main.c
Prog.profile
hello.c
clang
clang
main.opt.o
hello.opt.o
Linker
prog.opt.out


Deployment Workload
prog.opt.out
Deployment Output
Sample Profiler
Prog.sample.prof

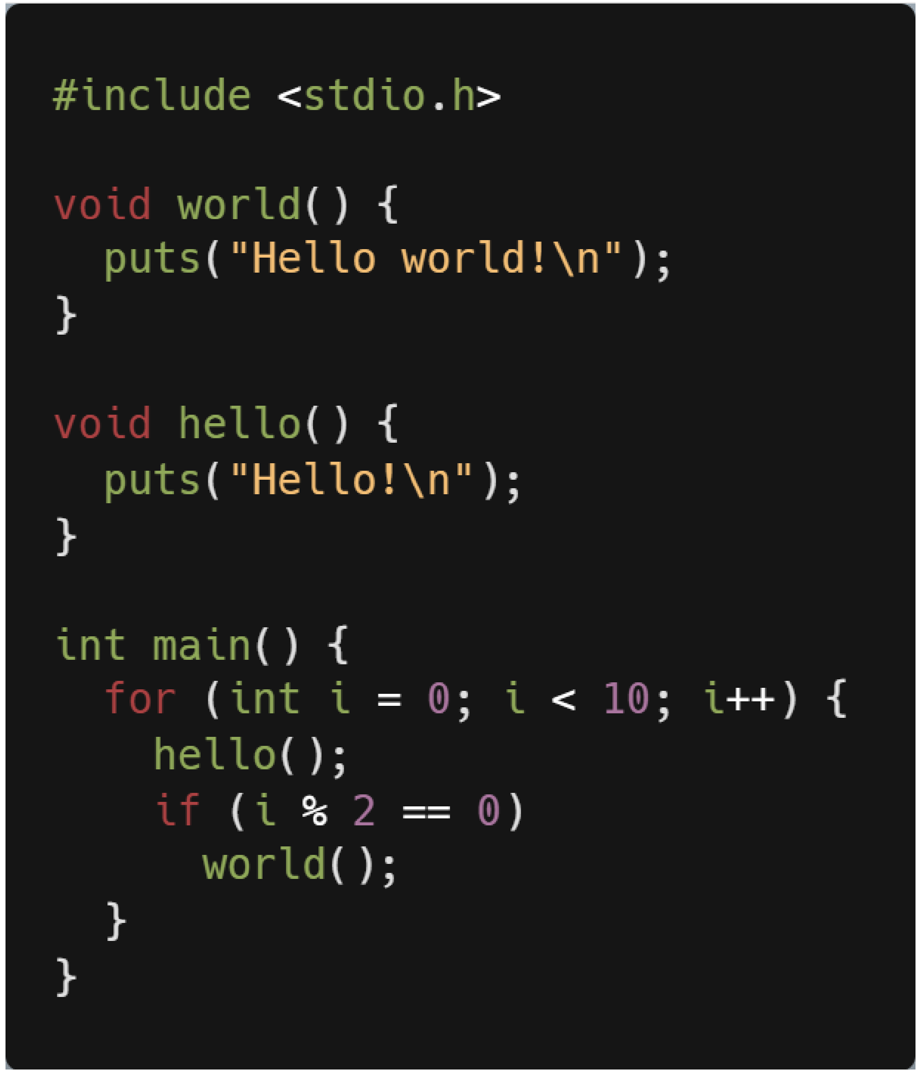

Prog.sample.prof
prog.opt.out
LLVM BOLT
prog.bolt.out

- That extra profiling and another optimization step adds a lot of complexity to the build and deployment pipeline.
- This extra complexity is not without benefits for BOLT’s design goals.
- But if we are just focused on binary layout improvement through function placement then there is a simpler approach.
- What information do we actually need to come up with a better function ordering?

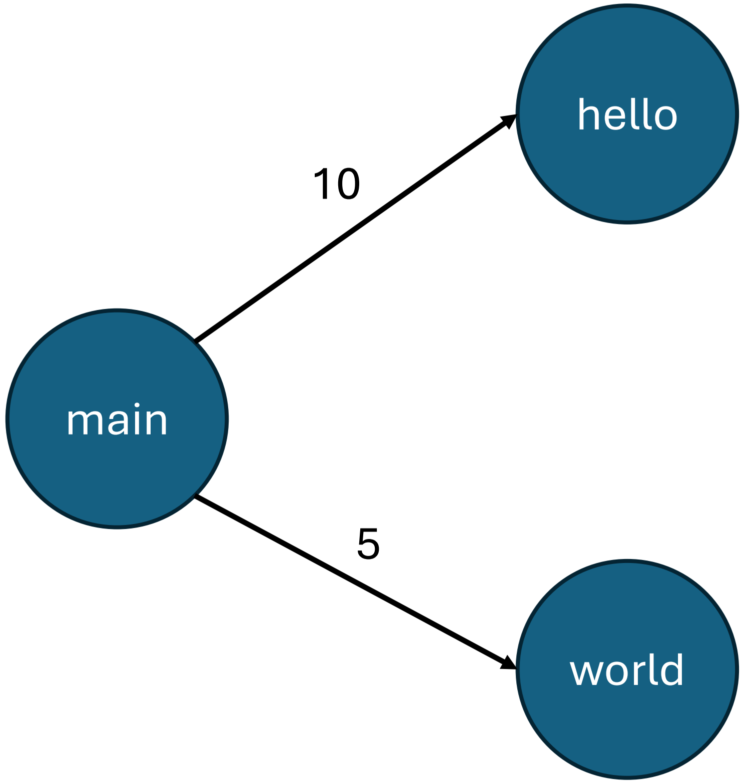
Function ordering problem formulation

- A call graph for an application is a directed weighted graph where the nodes represent functions in the application and the edges represent the caller-callee relationships.
- The weight of a given edge represents how many times that particular caller-callee relationship was exercised during a dynamic execution of a program.
- Note that the call graph depends on the dynamic execution of the program. In other words, the same program can have two different call graphs depending on program input.

Example call graph



```c
#include <stdio.h>

void world() {
  puts("Hello world!\n");
}

void hello() {
  puts("Hello!\n");
}

int main() {
  for (int i = 0; i < 10; i++) {
    hello();
    if (i % 2 == 0)
      world();
  }
}
```

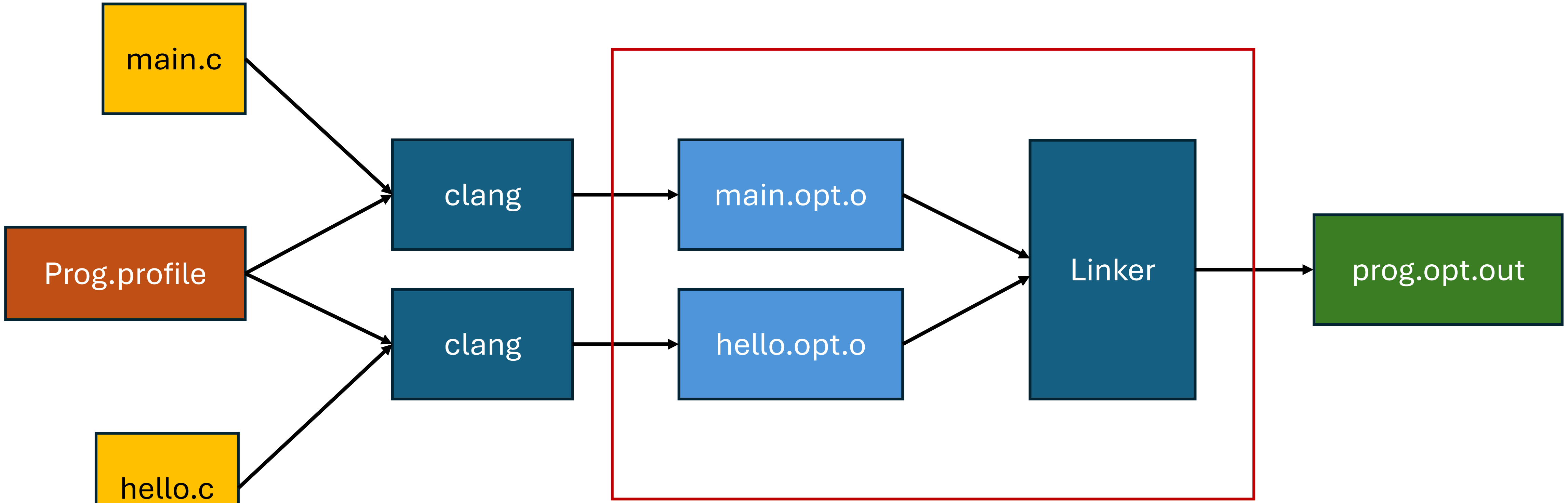


$$\begin{tikzcd} & \text{hello} \\ \text{main} \arrow[ur, "10"] \arrow[dr, "5"'] & \\ & \text{world} \end{tikzcd}$$

What now?

- Once we have the call graph, the problem of choosing a function ordering is can be formulated as choosing a sequence of nodes that minimizes some cost or using some other heuristic to choose which nodes should be adjacent in the ordering that we produce.
- For the purposes of this talk, we treat the actual algorithm as a black box. For more details on the actual algorithm, see the following papers.
- Pettis K., Hansen R.C., “Profile guided code positioning”, SIGPLAN Not. 25 (6) (1990) 16–27,.
- G. Ottoni and B. Maher, "Optimizing function placement for large-scale data-center applications," 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Austin, TX, USA, 2017, pp. 233-244, doi: 10.1109/CGO.2017.7863743.

Looking back at the PGO workflow



At this stage we have the information we need to produce a call graph.

Almost…

- Since we know how many times a given basic block is executed, if that BB contains a call instruction, we know how many times that call instruction was executed.
- The sum of execution counts of all call instructions in a given function to a callee is the weight of the edge from that function to the callee.
- The problem once again is that we only see a subset of a program at a time, therefore we never see the call graph for the entire program at once during the compilation process.

Solution

- We need to gather "partial call graphs" during compile steps and then later combine them all together to form a single call graph for the entire program.
- To achieve this, we embed the partial call graph from each compilation unit into its object file by serializing it writing it to the comment section of the produced XCOFF file.
- Then just before we link the binary together, we read in all the object files, extract the partial call graphs, resolve all the references and then pass our global call graph to our algorithm to get a function ordering then use it to choose a better layout.

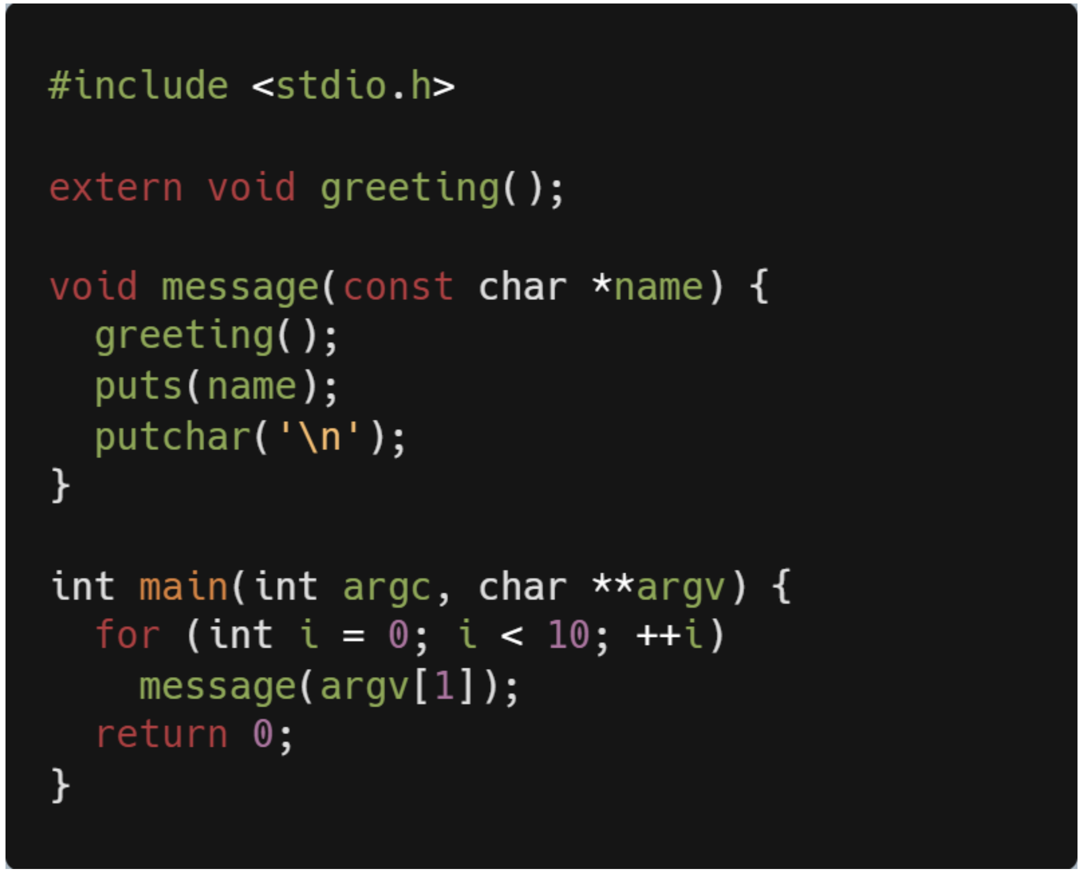
Partial call graph example

- Let's say we have the following two compilation units in our program.

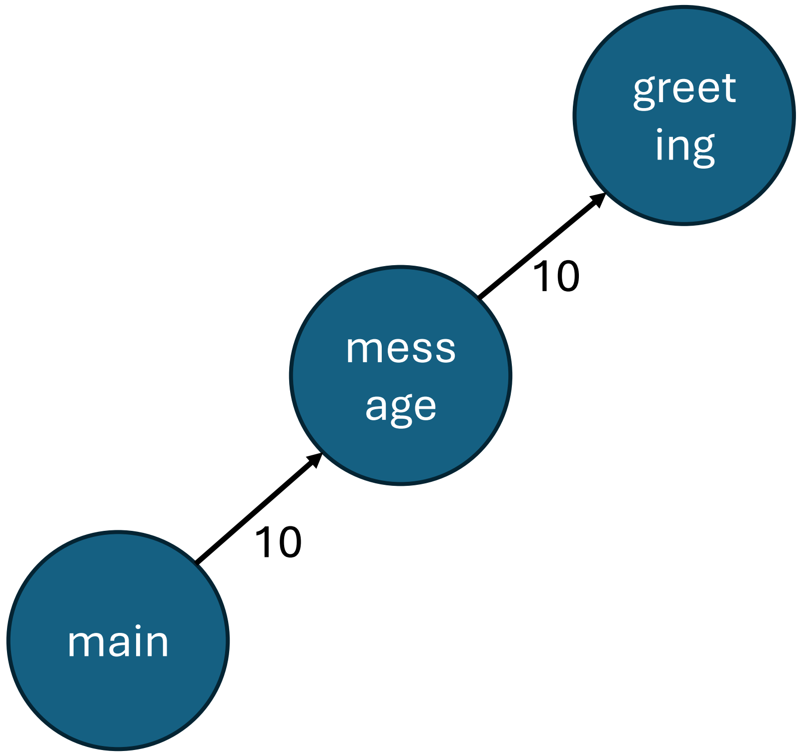
```c
#include <stdio.h>

extern void greeting();

void message(const char *name) {
  greeting();
  puts(name);
  putchar('\n');
}

int main(int argc, char **argv) {
  for (int i = 0; i < 10; ++i)
    message(argv[1]);
  return 0;
}
```

```c
#include <stdio.h>

void indent(int c) {
  for (int i = 0; i < c; ++i)
    putchar(' ');
}

void greeting() {
  indent(2);
  fputs("Hello", stdout);
}
```

Partial call graph for main.c

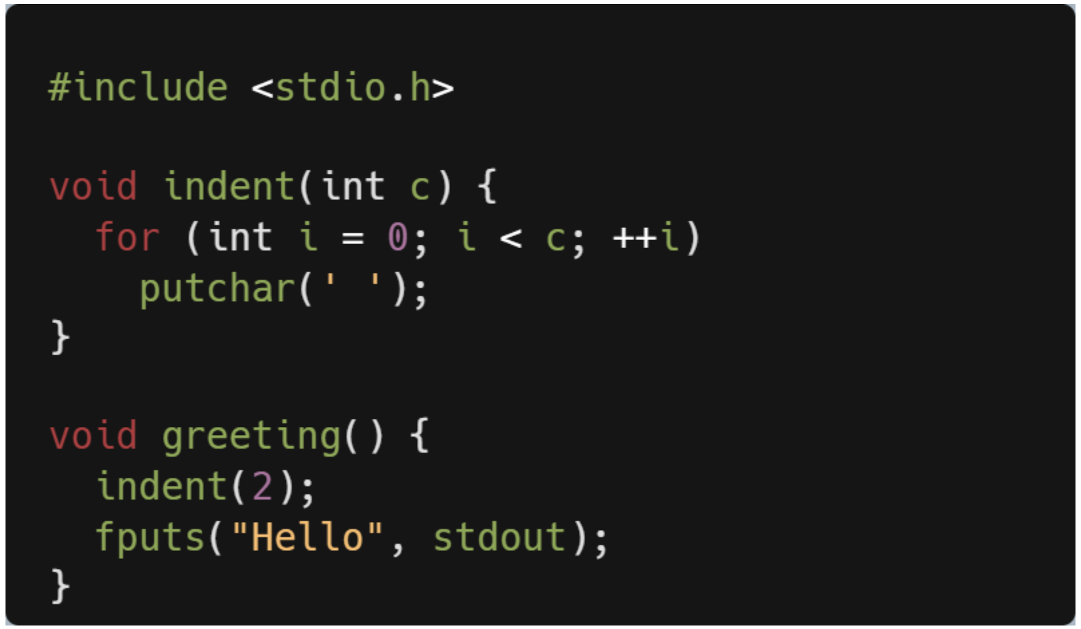


```c
#include <stdio.h>

extern void greeting();

void message(const char *name) {
  greeting();
  puts(name);
  putchar('\n');
}

int main(int argc, char **argv) {
  for (int i = 0; i < 10; ++i)
    message(argv[1]);
  return 0;
}
```

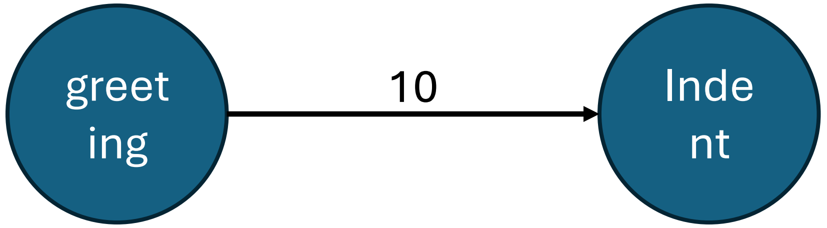


$$\begin{tikzcd} \text{main} \arrow[r, "10"] & \text{mess age} \arrow[r, "10"] & \text{greet ing} \end{tikzcd}$$

Partial call graph for hello.c

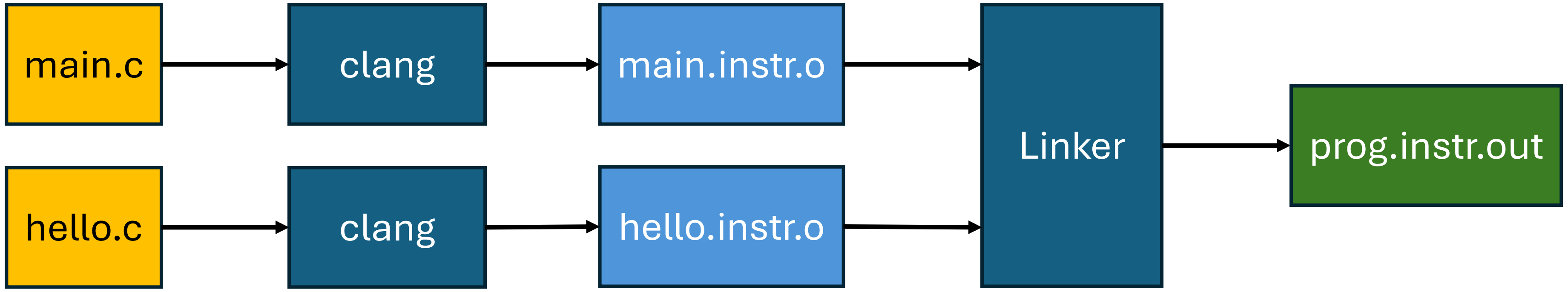


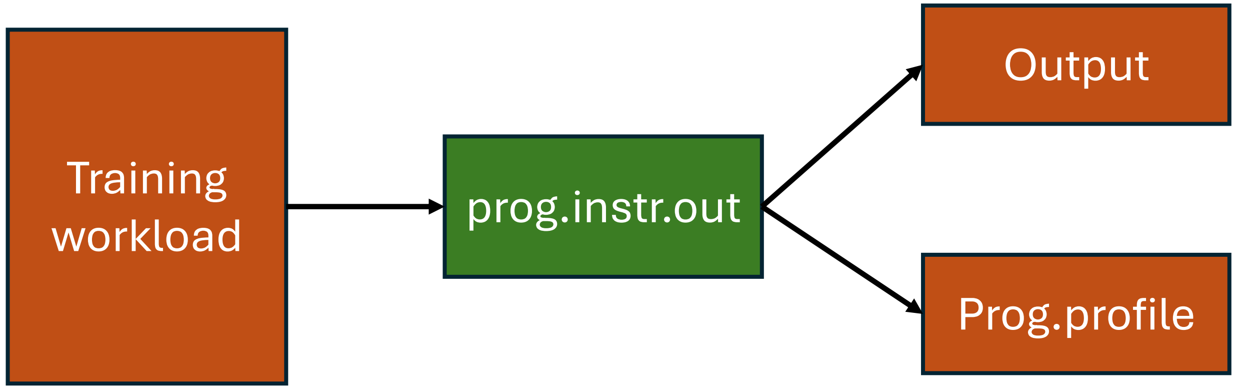
```c
#include <stdio.h>

void indent(int c) {
  for (int i = 0; i < c; ++i)
    putchar(' ');
}

void greeting() {
  indent(2);
  fputs("Hello", stdout);
}
```

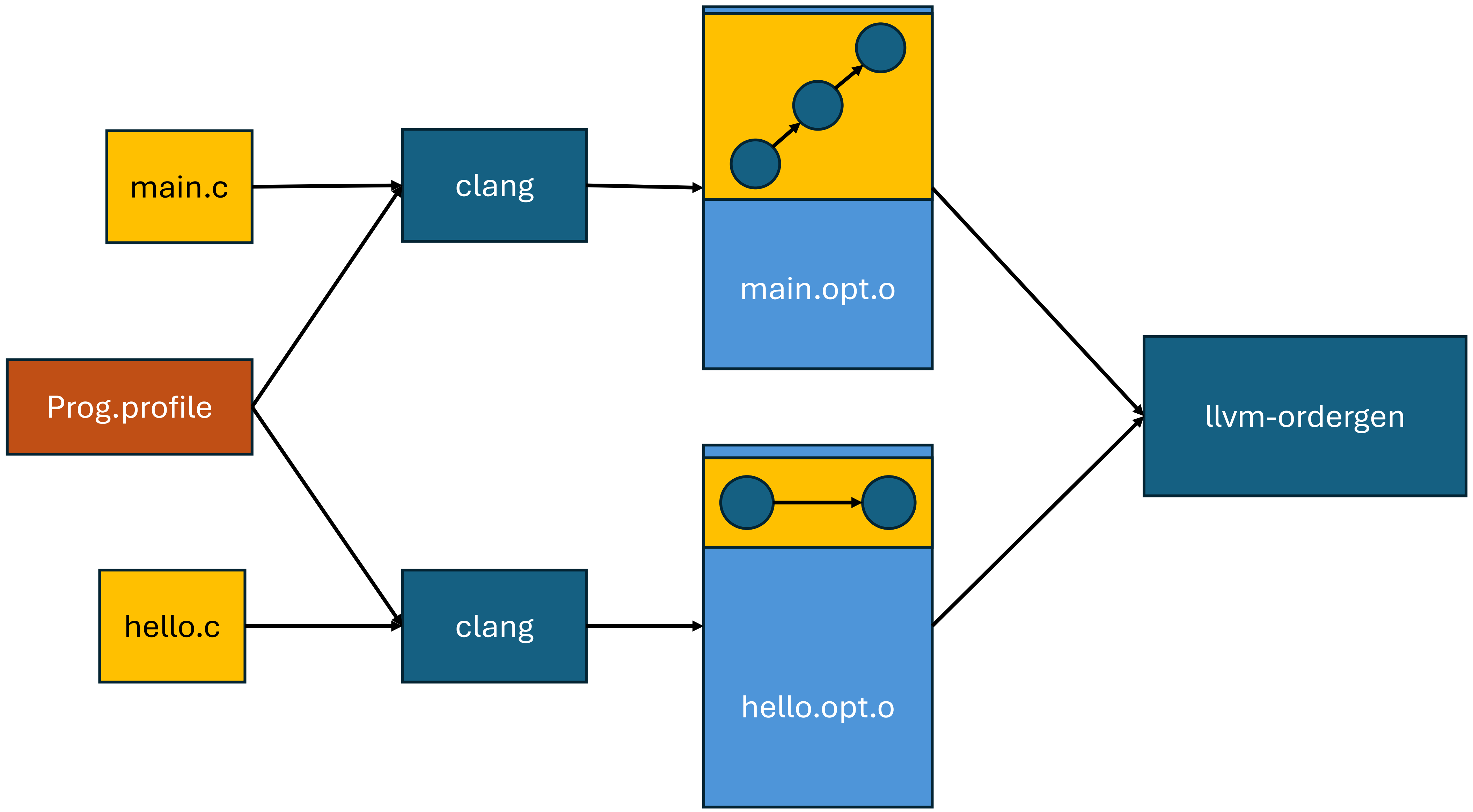


Workflow example

- We still need the first profile gathering step.

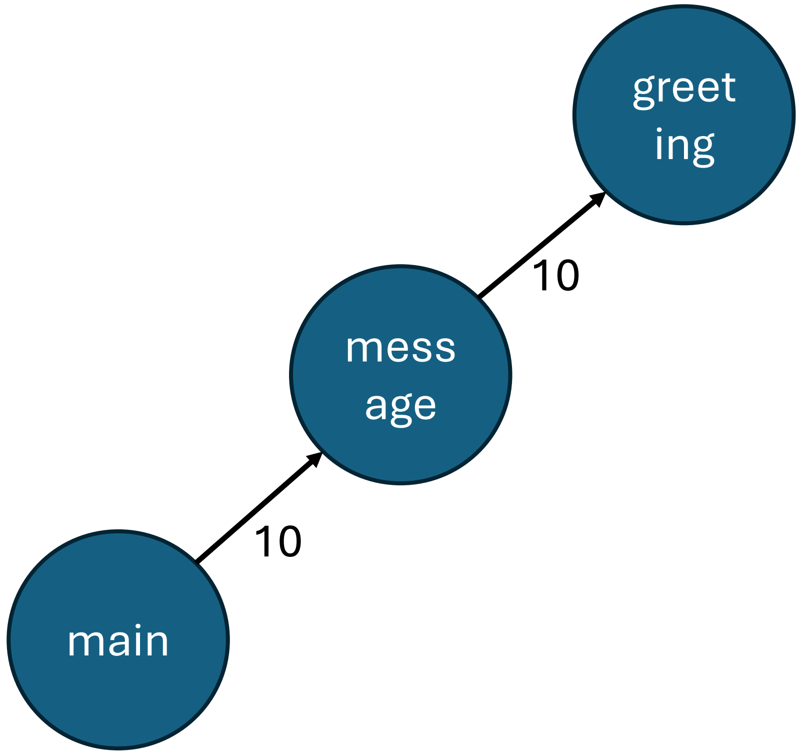


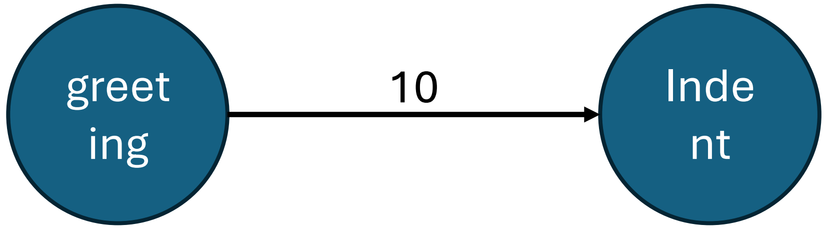



main.c
Prog.profile
hello.c
clang
clang
main.opt.o
hello.opt.o
llvm-ordergen

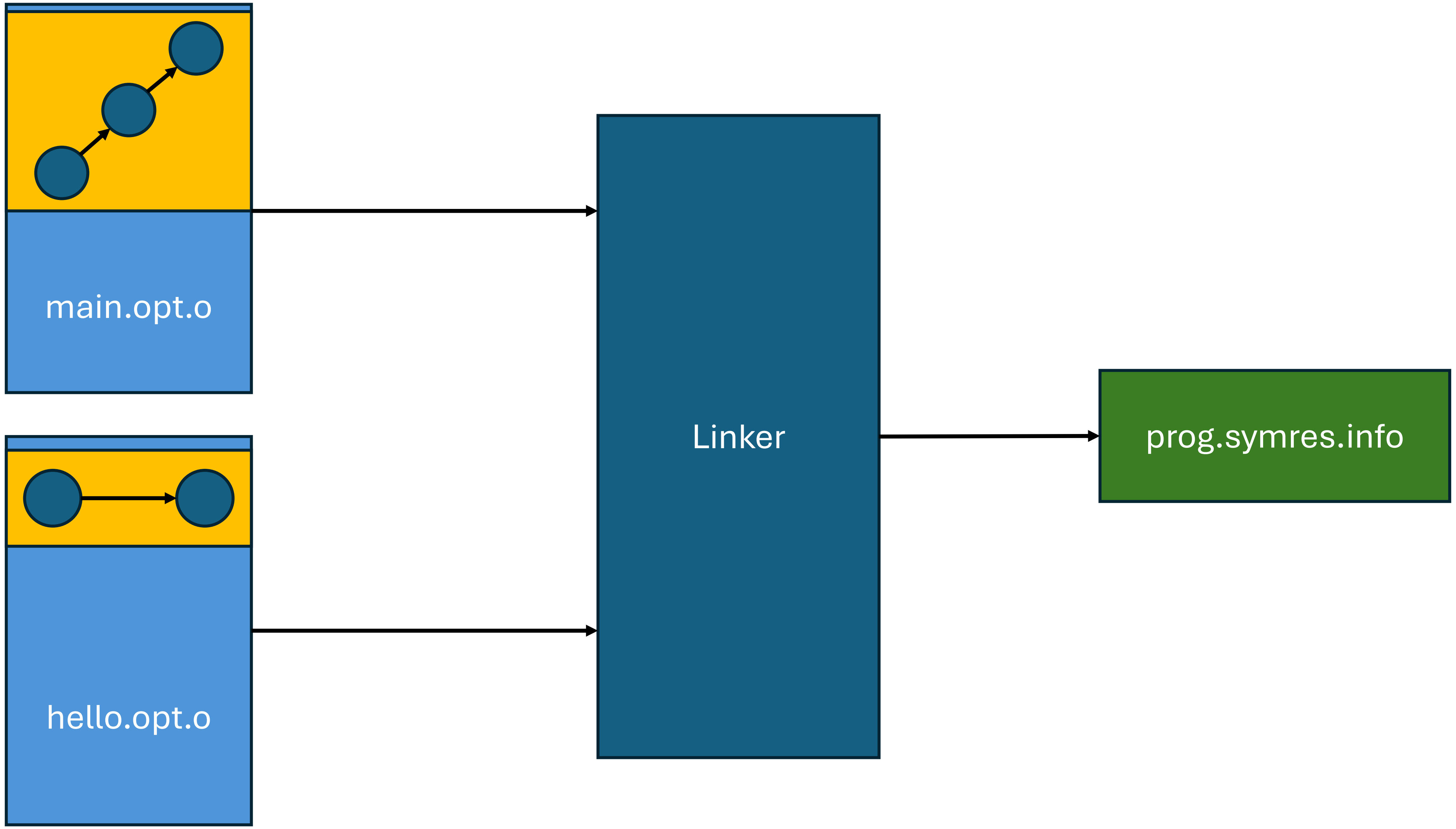
Inside llvm-ordergen

The task that llvm-ordergen must do at this point is pretty similar to that of a linker, i.e. symbol resolution

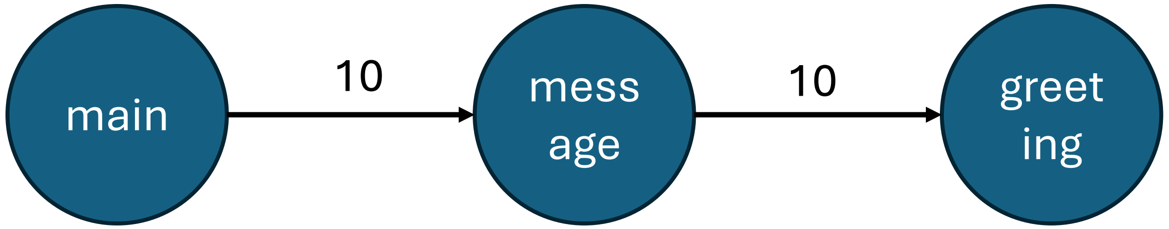


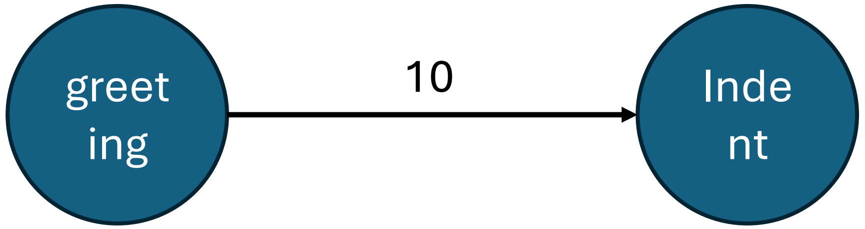


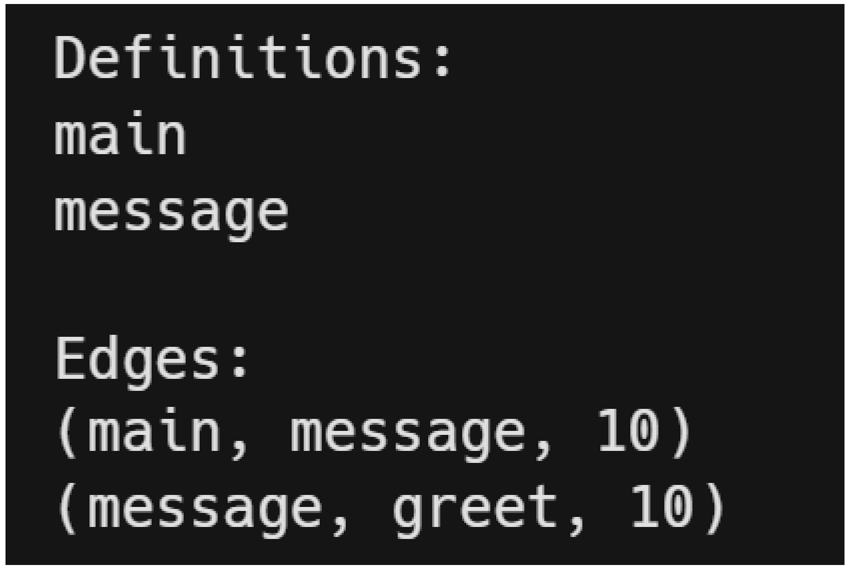
Getting symbol resolution info from the linker



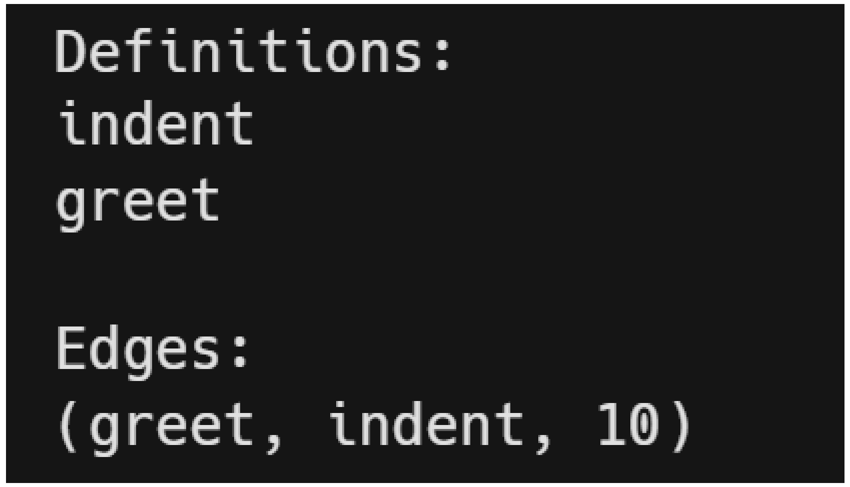
Inside llvm-ordergen



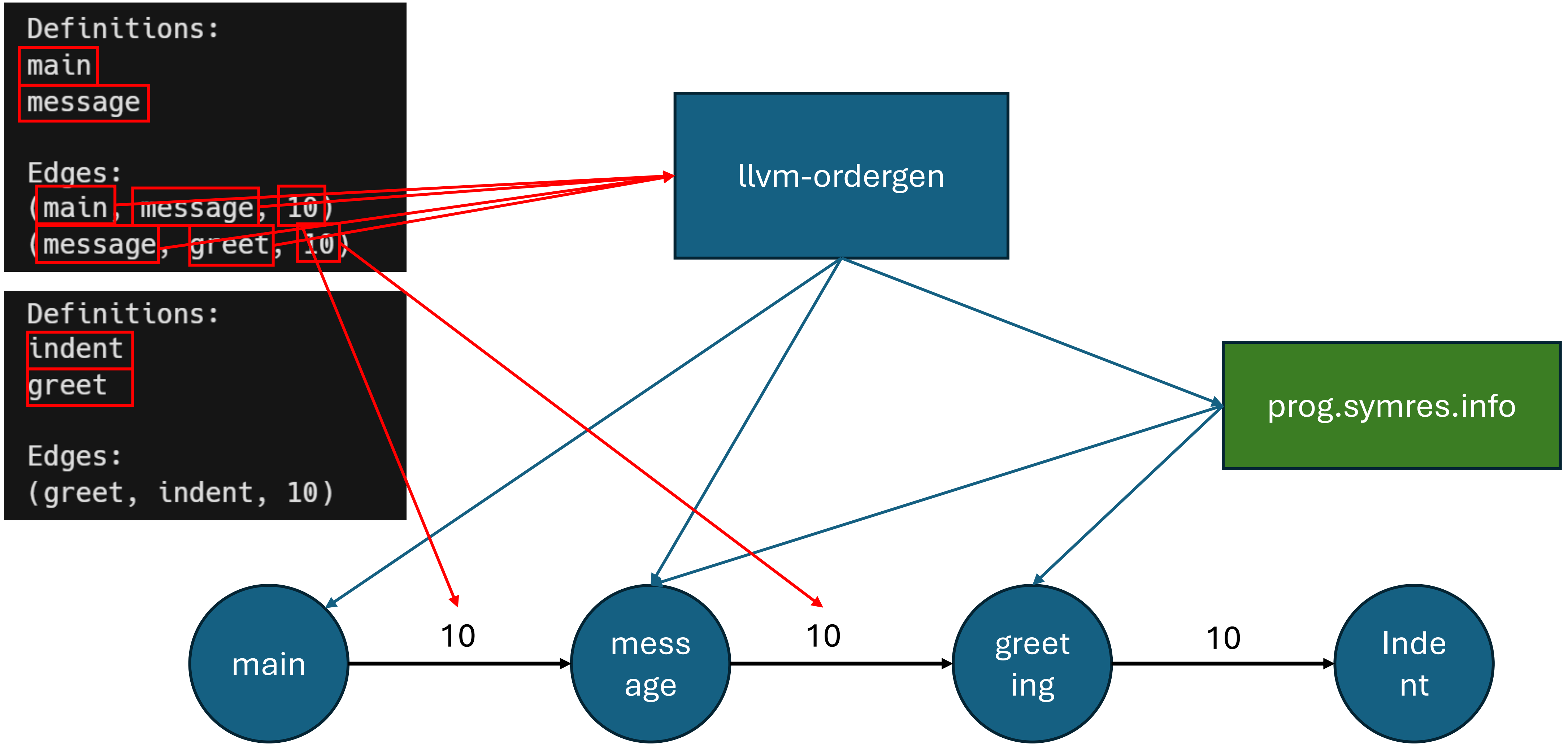




```
Definitions:
main
message

Edges:
(main, message, 10)
(message, greet, 10)
```

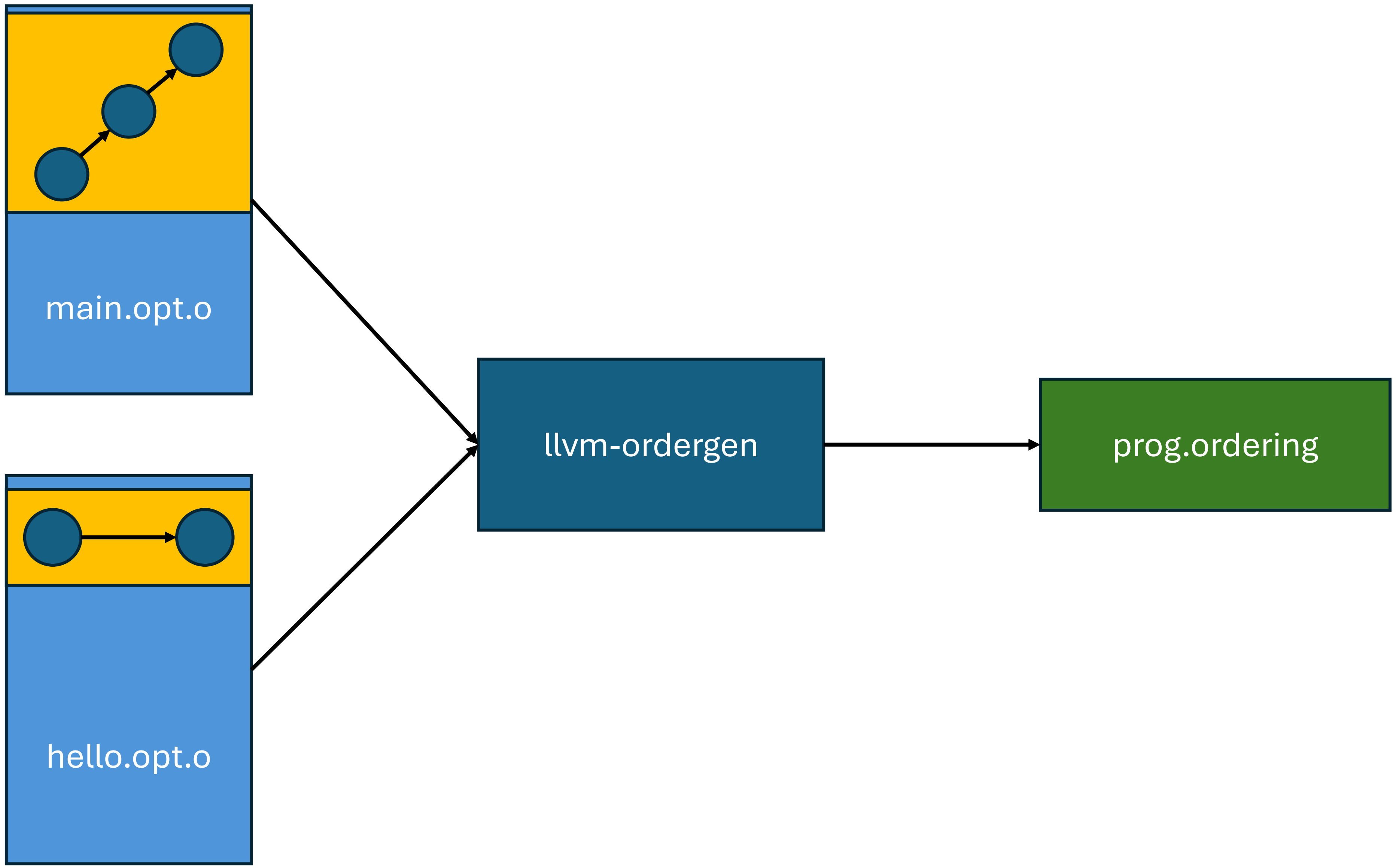


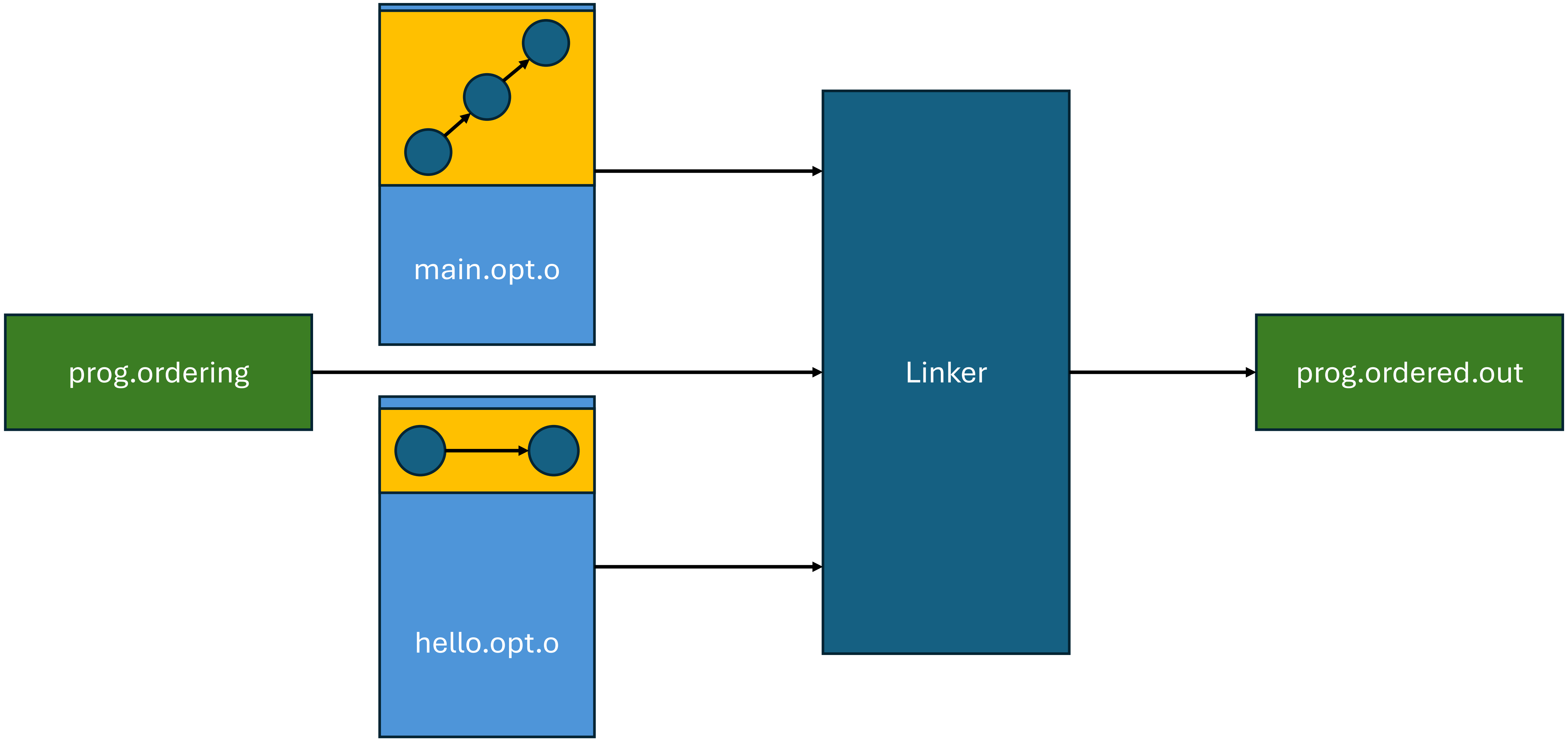
```
Definitions:
indent
greet

Edges:
(greet, indent, 10)
```

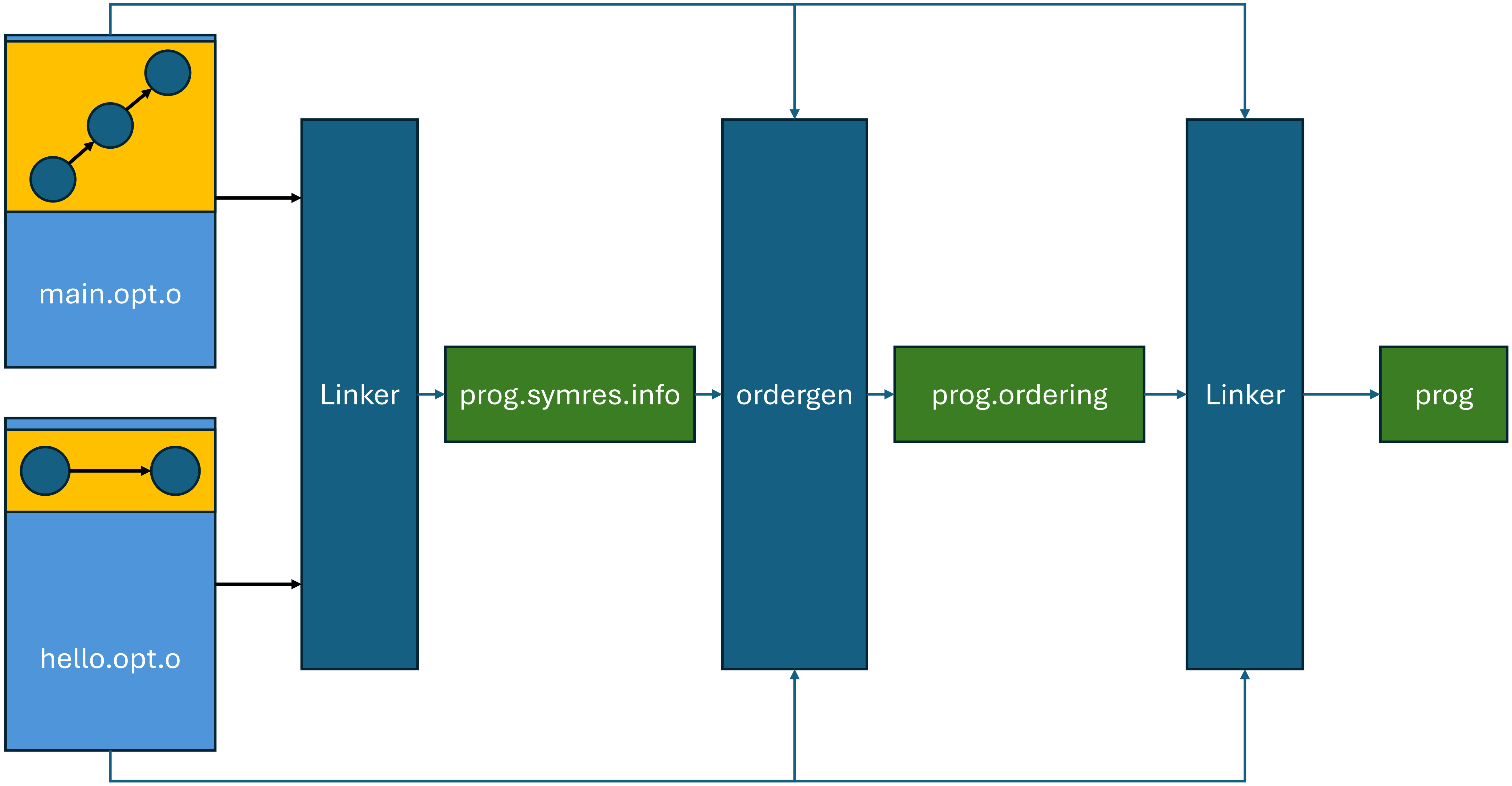


```
Definitions:
main
message

Edges:
(main, message, 10)
(message, greet, 10)
```

```
Definitions:
indent
greet

Edges:
(greet, indent, 10)
```


main.opt.o
hello.opt.o
llvm-ordergen
prog.ordering


main.opt.o
prog.ordering
hello.opt.o
Linker
prog.ordered.out


main.opt.o
hello.opt.o
Linker
prog.symres.info
ordergen
prog.ordering
Linker
prog

What does this solution give us?

- No extra profile step, we reuse the same profile information as a regular clang PGO build.
- The compiler driver can co-ordinate the order file generation process, making the feature completely transparent to the user/build system provided the program is linked through the compiler.
- For the user, the process is as simple as enabling a compiler flag.

Performance Evaluation

- Baseline: O3 with PGO
- Training workload: insensitive
- **3.6%** performance improvement over the baseline.
- Improvements in the compiler frontend.

Thank you.

Dhanrajbir Singh Hira
University of Alberta