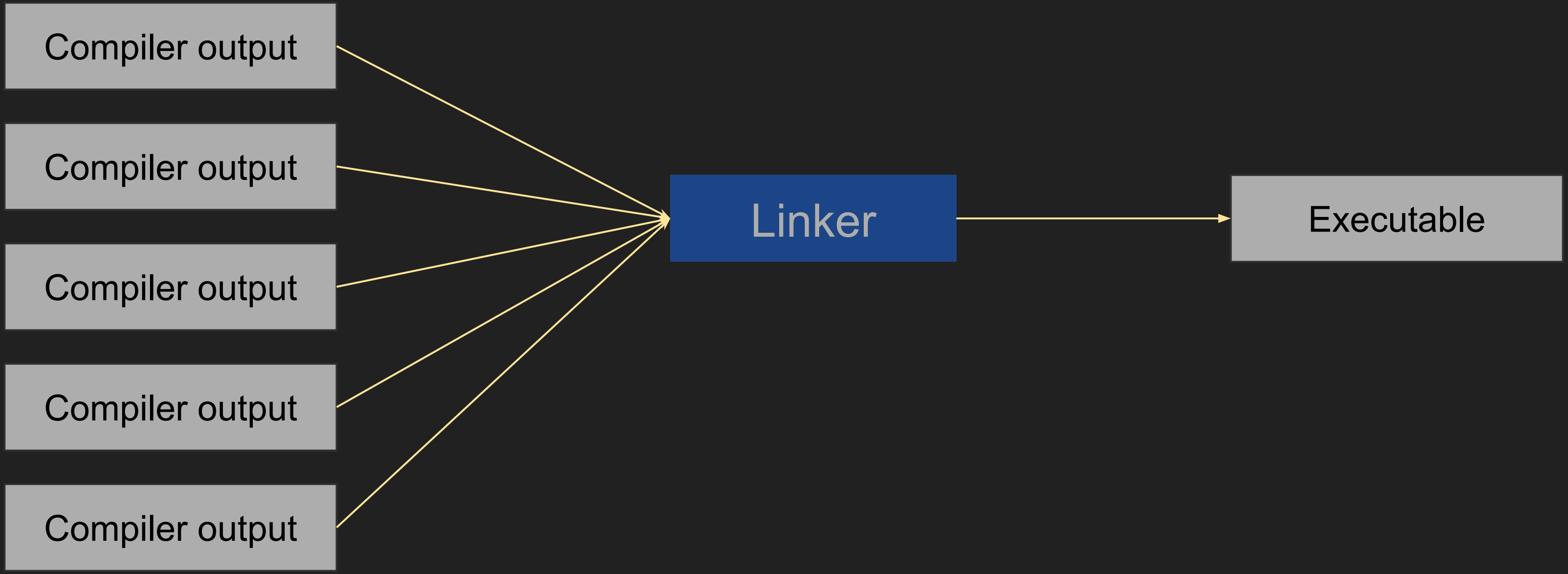
A linker in the Wild

David Lattimore

Talk outline

- High level overview of what a linker does
- Motivation for writing a linker
- More details about some things linkers need to handle
- Tools and techniques for debugging the linker
- Implementation details
- Current status
- Performance
- How this can speed up Rust build times

What a linker does



Compiler output

Compiler output

Compiler output

Compiler output

Compiler output

Linker

Executable

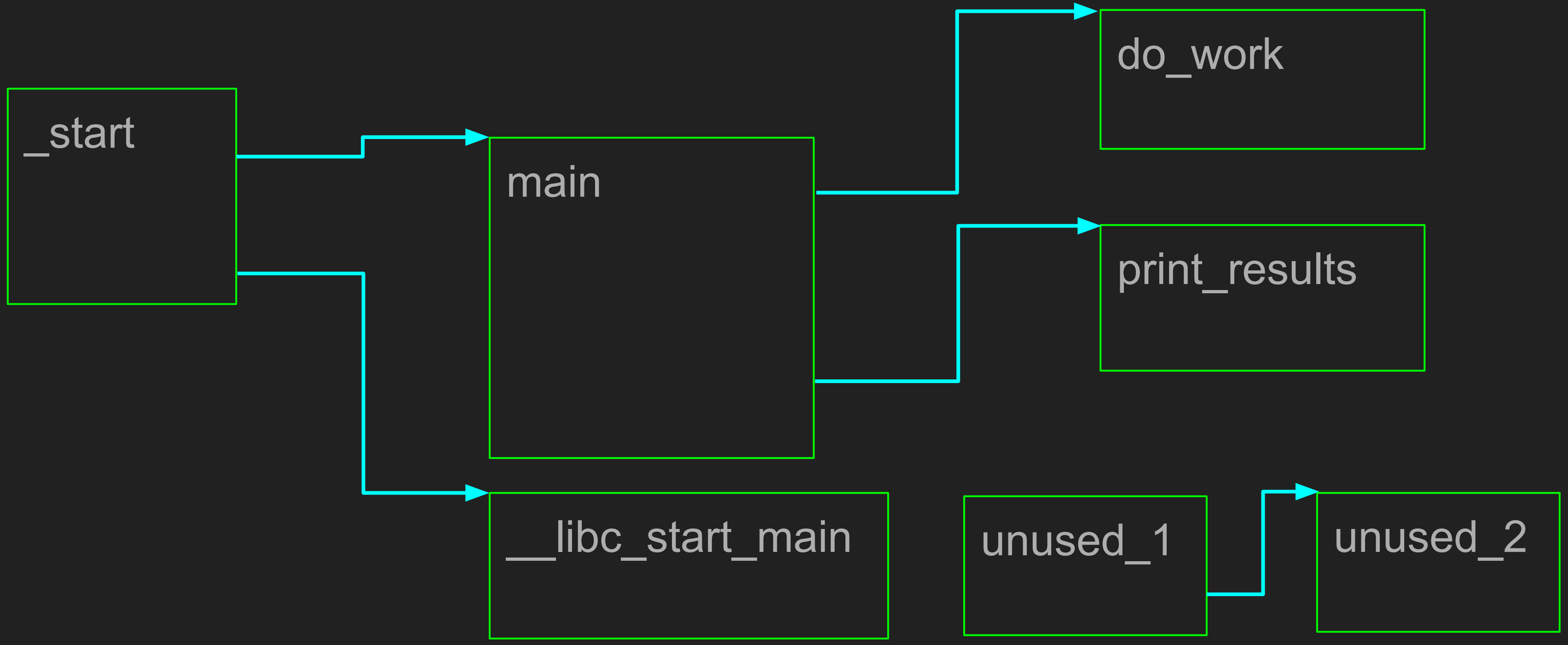
Why another linker?

- Existing linkers: GNU ld, gold, lld, mold
- For incremental builds, linking is often the slowest stage
- Tighter integration with rustc could allow speeding up cold builds too

Section types

- Executable
- Read-only
- Read-write
- Read-write zero-initialised
- Many specialised section types generated or handled specially by the linker
    - Global offset tables (GOT)
    - Procedure linkage table (PLT)
    - Symbol tables
    - Dynamic symbol tables

Input sections and relocations



Thread local storage

- Declared in Rust using the thread_local! macro
- Each thread has a separate copy of the variable
- Access models:
    - Global dynamic
        - 1 function call per access
    - Local dynamic
        - 1 function call to access multiple variables
    - Initial exec
        - 2 instructions per access + 1 runtime relocation per variable
    - Local exec
        - 1 instructions per access
- Linkers transform slower access models to faster ones when possible

ifuncs

- Several versions of a function + a resolver that picks which to use
- Resolver gets called at program startup and is passed CPU information
- Linker needs to write relocations that glibc uses at startup to resolve these

Exception frames (eh_frames)

- Used at runtime for stack unwinding including backtraces and panics
- Related to debug info, but more limited
- Frame information for each function in your binary
- Linker discards frame information for functions that are garbage collected.
- Linker needs to build a binary search index

Many more things linkers need to handle

- String merging
- Common symbols: zero-initialised. Size is the maximum of possibly multiple definitions.
- Custom sections with start/stop symbols
- Weak symbols
- Archive semantics

Debugging linker errors

- objdump and readelf to examine the binary output
- gdb for stepping through the program to see where it’s going wrong
- Usually you’re just looking at assembly code
- Step through the two binaries at once - one from our linker, one from the default system linker
- rr is very valuable - lets you step backward
- Working on an ELF diff tool

Implementation details

- Most phases make heavy use of threading, mostly using Rayon
- Memory–maps input and output files
- Currently only use of unsafe is mmap - which is unfortunately unsound if an input file is changed while we run, but there’s nothing that can be done about that on Linux
- Avoids heap allocation as much as possible

Current status

- Currently Linux on x86-64 only
- Static linking works reasonably
  - Tested with glibc and musl
  - Non-relocatable and position independent
- Stack unwinding works
- Dynamic linking still very newly implemented
  - Only tested on trivial programs so far
- Output to shared objects (e.g. proc macros) is still a work-in-progress
- Doesn’t yet support debug info

Current performance

Benchmark for a medium-sized, statically linked binary

Linker	Time (ms)
GNU ld	12300
Gold	3365
lld	905
mold	457
wild	363

Incremental linking

- Not yet started, but that’s what the “I” in “Wild” stands for
- Avoid repeating the same work each time you do an incremental build
- If you’ve only added a print statement to one function, you shouldn’t need to relink your entire binary.
- Definitely only intended for development purposes
- Output won’t be bit-identical to what you’d get when linking from scratch

Future work: Rustc integration

- Either:
  - Build the linker into the Rust compiler; or
  - Load part of the Rust compiler into the linker via some plugin mechanism
- Allows optionally deferring the following to link time:
  - Codegen
  - Monomorphisation
  - Inlining

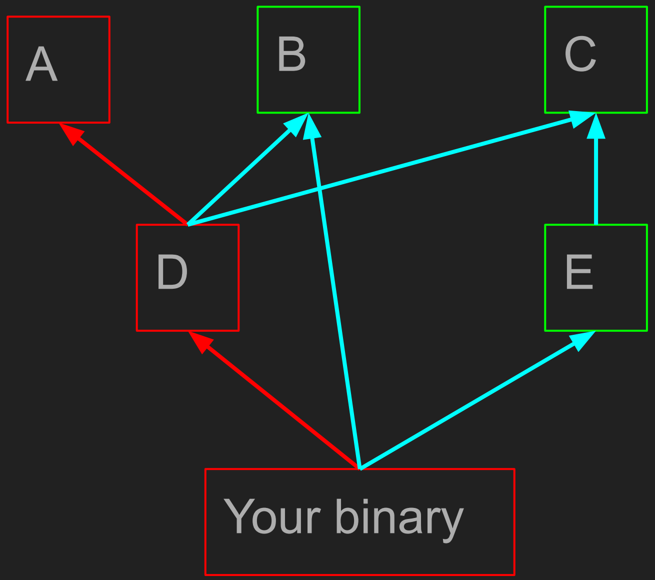
Avoiding codegen of dead code

- Optionally defer codegen to link time
- Allows us to take advantage of the linker’s garbage collection
- Caching needed

Avoiding repeated monomorphisation

- Monomorphisation makes generic functions concrete by substituting generic parameters for concrete types.
- Vec::<T>::push → Vec::<u32>::push
- Each codegen unit that calls Vec::<u32>::push repeats this monomorphisation.
- Optionally defer monomorphisation to link time

Avoiding recompiles when editing dependencies



Summary

- Incremental builds can be sped up by
  - Incremental linking
  - Deferring inlining to link time
- Cold builds can be sped up by
  - Deferring codegen to link time
  - Deferring mononorphisation to link time

Thanks to my github sponsors!

- repi
- Urgau
- joshtriplett
- acshi
- teh
- bes
- yerke
- alexkirsz
- Pratyush
- lexara-prime-ai
- ethanmsl
- +1 anonymous