A linker in the Wild

David Lattimore

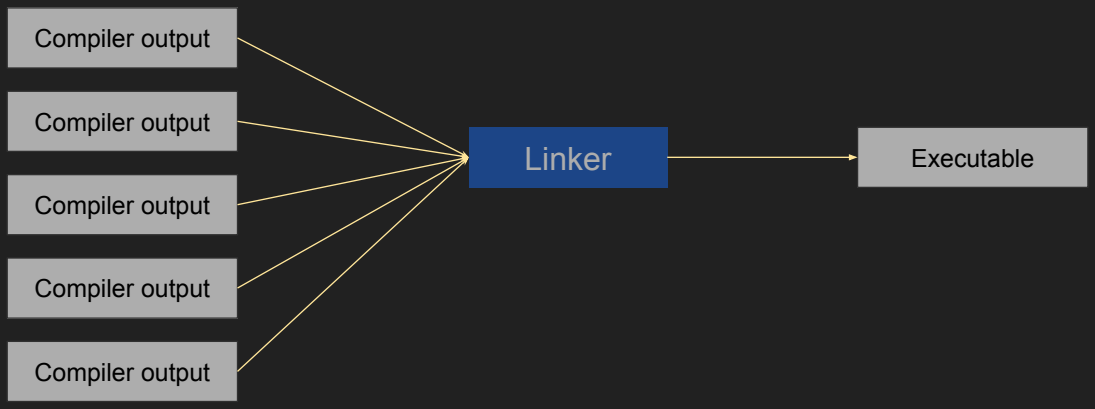
For the past few months I've been developing a linker in Rust. The linker is called Wild, which is a recursive acronym that stands for Wild Incremental Linker. It's not yet incremental, but that is the eventual plan.

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

I'll start by giving a very high level overview of what a linker does before discussing the motivations for writing a linker. I'll then go into some more details about some of the weird things that linkers need to handle. I'll discuss some of the tools and techniques I use when debugging the linker. I'll go over a few implementation details, then talk about the current status and performance of the linker, and finish with some ideas for future work.

What a linker does



At a very high level, a linker's job is to take multiple compiler outputs and merge them together into a single binary. As we'll see though, there are many details that make this much more complicated than it might at first appear.

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

There are already several open source linkers available on Linux. Specifically GNU ld, gold, lld and mold. With four existing high quality linkers available, why write yet another one? My main motivation is to speed up Rust compile times, in particular incremental compilation times.

With the right settings, for small programs, Rust compile times are generally pretty good. However as the size of your code and in particular the size of your dependencies grows, compilation speeds can become substantial. Because linking is done from scratch each time you rebuild, linking is often the slowest stage of incremental compilation.

An incremental linker could help here, by avoiding the need to redo lots of time-consuming work every time you rebuild.

Furthermore, tighter integration between the Rust compiler and the linker could help speed up cold build times as well.

Section types

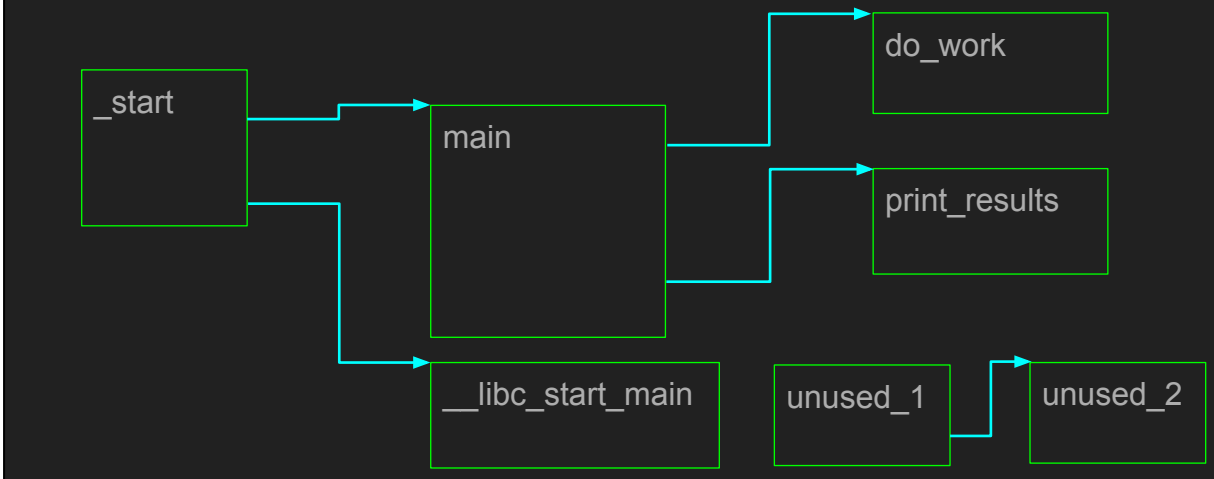
- Executable
- Read-only
- Read-write
- Read-write zero-initialised
- Many specialised section types generated or handled specially by the linker
  - Global offset tables (GOT)
  - Procedure linkage table (PLT)
  - Symbol tables
  - Dynamic symbol tables

I'll now go into a little more detail about what linkers do.

The input files to the linker don't just contain executable code. They also contain other kinds of sections including read-only data, read-write data and zero-initialised read-write data. The latter is special in that because it's all zeros, it doesn't need to take up any space in the file. The linker does however still need to lay out this section.

There are also quite a few specialised section types that either need special handling by the linker, or are created by the linker. For example there are global offset tables, which contain pointers to functions or data that generally aren't resolved until runtime. There are procedure linkage tables that contain little bits of machine code generated by the linker that jump to functions via the global offset table. Symbol tables are useful for debugging and giving good stack traces. Dynamic symbol tables are required for resolving symbols at runtime in dynamically linked executables and shared objects.

Input sections and relocations



Each input file is composed of multiple sections. Each section may then have some relocations specified. Relocations are references to other sections or symbols and are instructions to the linker to fill in some address or value at a particular offset within the section. In the diagram here, the green boxes are sections and the blue lines are relocations.

Depending on how the code being linked was compiled, there may be multiple functions or variables per section. However, more common is to have one function or variable per section. This allows the linker to perform a garbage collection phase, where it traverses the graph of input sections starting from some roots and then discards sections that aren't reachable from those roots. So in this example, `unused_1` and `unused_2` aren't reachable, so can be discarded. Some sections will have flags on them to tell the linker that the section should not be discarded even if it's not reachable. These are then used as roots by the garbage collection phase.

On Linux there are about 40 different kinds of relocations. I'll only look at a few especially interesting ones, which are related to thread locals.

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

Rust lets you declare thread local variables using the thread_local! macro. Each thread then has a separate copy of this variable.

Thread local variables have several different access models. The most general and also the slowest is global dynamic. This might be used where you have a shared object such as a plugin that was loaded at runtime and needs to access a thread-local that is declared by the main program or another shared object. A common example of such a thread-local is C's errno variable.

Local dynamic is similar to global dynamic, but is used when a thread local is accessed from the same shared object that declared it.

Initial exec can be used between the main program and shared objects, but only for shared objects that were loaded at startup.

Local exec is the fastest and applies to thread-locals that are declared and used within the code in the main binary - i.e. not from shared objects. Reading or writing a thread-local with the local exec model can be done in a single instruction.

Each of these access models has separate relocation types that the linker needs to handle. Additionally, library code is often compiled with the most general access model, since we may not know at compile time whether the code is going into a shared object or into a binary. The linker however does know, so it can transform code compiled say with the global dynamic model into the local exec model.

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

Another interesting feature is ifuncs, which are a way to provide several versions of a function, each optimised for different CPU features. At program startup, glibc calls a resolver function, passing it information about the current CPU. The resolver function returns a pointer to the selected implementation for the current CPU. The linker needs to output relocations for all ifuncs so that glibc can find and resolve them at program startup.

An interesting aside is that the recent xz backdoor used an ifunc resolver to run code that installed the backdoor, but more commonly they're used to provide faster versions of functions like memcpy.

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

Exception frames are needed in order to produce backtraces or perform stack unwinding, for example if your code panics.

Each function in your binary needs frame information. The compiler, somewhat unfortunately puts all of these into a single section, so the linker has to split this apart and discard frame information for any functions that were discarded during the garbage collection phase. The linker also needs to build a binary search index which is used by the runtime to locate the frame information for an arbitrary address without having to scan through every frame.

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

There are many more things that linkers need to handle. I'll briefly go through a few of them here.

C programs often repeat the same string in many compilation units. When those strings are put into a string-merge section, the linker is responsible for deduplicating them.

Common symbols are a feature that originally came from Fortran, but for some reason was ported to C and is used a little bit by glibc. A common symbol is zero-initialised, however can be declared multiple times with different sizes. The linker is responsible for determining the maximum size of any declaration and using that.

Input files can define custom sections. The linker needs to merge all custom input sections with the same name together. The linker also defines symbols that point to the start and stop of the custom section so that user code can find everything that went into the section at runtime.

Symbols can be marked as weak, allowing them to be overridden by strong symbols with the same name. It's also OK to have a weak undefined symbol that is never defined, in which case the weak symbol resolves to the value 0. i.e. the null pointer.

Archives are a file format that is used to bundle several object files together. However linkers treat objects within archives differently than objects not in archives. Specifically an object in an archive is completely ignored unless it defines a symbol that a

previous object listed as undefined. Another annoying thing about archives is that entries are only 2-byte aligned, which means we need to be able to handle objects in archives that don't have 8-byte alignment. This is especially relevant to Rust since the Rust compiler outputs archives for all the code that it compiles.

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

When the linker gets something wrong, it can often be pretty tricky to figure out what the problem is.

Two very useful tools are objdump and readelf. These let you examine all the headers and sections within an ELF file. There's quite a bit of overlap between these tools, but often one tool will give a little more information than the other.

For debugging the linker output, I've mostly used gdb to step through the program. Usually you're just stepping through the assembly code, since that's closest to what the linker is working with.

The equivalent binary produced by GNU ld can be used as a reference to compare to - either in a tool like objdump or by stepping through both binaries in two separate gdb sessions to see where they diverge.

Mozilla's rr tool is a replay debugger which is incredibly useful. Whenever I used gdb, it's almost always via rr. This lets me step the program backward from the point where something went wrong to see how it got there.

I've also started work on a tool for finding differences in ELF binaries even when they have a completely different layout.

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

I'll now go through a few implementation details of Wild.

Most phases of the linker are multithreaded. Where possible I use Rayon's par_iter for this. However, there are two graph algorithms in the linker - one for determining which archived objects to load and one for garbage collection. These don't fit so well into Rayon, so have more manual thread implementations.

The input files are memory-mapped, as is the output file. This means that it's crucial that the input files not get changed while the linker is running, otherwise we'll get undefined behaviour, most likely resulting in a crash. Other linkers do the same thing, as it's necessary for good performance. It's unfortunate that Linux doesn't provide any truly safe way to use mmap.

Wild avoids heap allocation as much as possible. Where heap allocations are needed, at least some of them are large heap allocations that are then split into parts and shared between multiple threads. In order to reduce heap allocation and also to reduce copying, we borrow data from the input files wherever possible. For example the hashmap used to look up symbols by name borrows the names from input files rather than copying them onto the heap. Each phase of the linker borrows data from the previous phases. Rust's scoped threads have proved very valuable for making this work safely.

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

Wild so far only supports Linux on x86-64, although I'd like to expand this in the future.

Static linking with glibc and musl works reasonably well, including position-independent binaries. Stack unwinding also works.

Basic dynamic linking has been implemented, but there's still more to be done.

Output to shared objects such as used by proc macros is still a work in progress and I don'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

Performance wise, I've only done limited benchmarking. This slide shows the times for each linker to produce an 80 MiB, statically linked, non-relocatable binary with no debug info.

GNU ld is the default linker on most Linux distributions and is also by far the slowest. On this benchmark it takes more than 12 seconds. The gold linker is a good bit faster at about 3 seconds. LLD takes 900 ms, Mold about 450 ms and Wild about 360 ms.

I've tried to make this test as fair as I can by only using features supported by all the linkers. But there's certainly still the possibility that as I do bug fixes and feature work on Wild, that it will slow down. The goal though is to eventually be incremental, which is where the real speed boost will come from.

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

I haven't yet started work on making Wild incremental, since I want to get it reasonably feature-complete first. I am somewhat committed to it though, given that the I in Wild stands for incremental.

The idea of incremental linking is to avoid repeating the same work each time you do an incremental build. This means that if you've only added a print statement to one function, you shouldn't need to relink your entire binary.

It's worth mentioning that incremental linking would only ever be intended for development purposes. For a release build, you'd definitely want to link from scratch. One reason for this is that release builds should be reproducible while incremental link outputs won't be bit-identical. They'll also be larger, since we'll need to leave space in the binary for sections to grow as the code is edited.

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

One area where I'm especially excited to have a linker written in Rust is to explore tighter integration between the rust compiler and the linker. This might take the form of building the linker into rustc, or having the linker load parts of rustc as some kind of linker plugin. Either way, the idea would be to defer some stages of the compiler to link-time. In particular codegen, monomorphisation and inlining. I'll go through these on the next slides.

Avoiding codegen of dead code

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

Codegen is the part of compilation that converts your code into machine code that is executable by the processor. It's also where most of the optimisations happen and is one of the more expensive parts of compilation.

The idea of moving it to link time is to take advantage of the linker's garbage collection phase, which knows which bits of code are needed for the binary and which aren't. By deferring codegen to link time, we can thus avoid doing codgen for functions that are dead code. This might be code in dependencies that are only partially used, or things like Debug impls that you are only using when debugging stuff, but aren't called normally.

Caching would be needed to ensure we don't need to codegen the same code if we're building multiple binaries.

Avoiding repeated monomorphisation

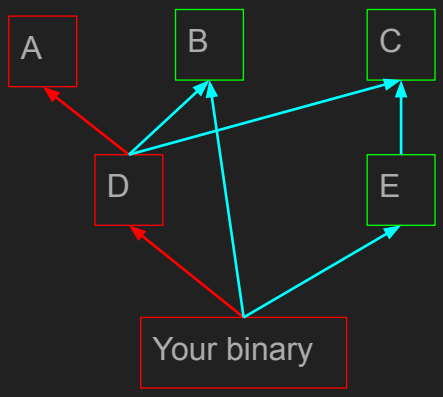
- Monomorphisation makes generic functions concrete by substituting generic parameters for concrete types.
- Vec::<T>::push → Vec::<u32>::push
- Each codegen unit that calls Vec::<u32>::push repeats this monomorphisation.
- Optionally defer monomorphisation to link time

Monomorphisation is when the compiler converts generic functions into regular functions by substituting concrete types. For example, when Vec::<T>::push is converted to Vec::<u32>::push.

Each codegen using that calls Vec::<u32>::push repeats this monomorphisation and perhaps more importantly the associated codegen. Repeating the same work is wasteful.

By moving monomorphisation to link time, we can avoid this repeated work, ensuring that we only monomorphise each version of Vec::push once.

Avoiding recompiles when editing dependencies



Another common problem that causes slow compiles is shown in this diagram. If you're making edits to crate A, then rebuilding your binary, currently that requires recompiling A, D and your binary. If your edit to A didn't change the interfaces provided by A, then in theory, it should be possible to only recompile A, then relink your binary.

However when D was compiled, the function that we edited in A might have been inlined, so we currently need to recompile D as well. Rerunning the compiler on the crate D just so that we can redo one inlined function is a waste. even with incremental compilation, we still redo lots of work. If we instead defer inlining to link time, we can just redo the inlining that changed in the linker and can avoid rerunning the compiler on D.

This change effectively means that we'd only need to rerun rustc on crates that changed, and when interfaces were changed, on crates that made use of those interfaces.

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

My goal with this project is faster rust compile times, both incremental and cold builds.

Incremental builds can be sped up by incremental linking and by deferring inlining to link time.

Cold builds can be sped up by deferring codegen and monomorphisation to link time.

I think having a linker written in Rust has great potential to help speed up Rust compilation.

Thanks to my github sponsors!

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

I'd like to finish by saying a big thank you to all of my github sponsors for helping make it possible for me to work on this full time. Does anyone have any questions?