Pass Plugins

Past // Present // Future

Stefan Gränitz EuroLLVM Berlin 15 April 2025

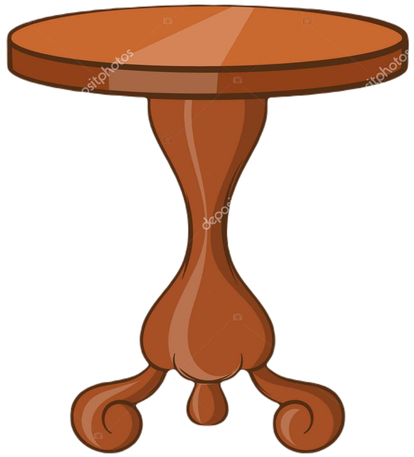
Agenda

1. Past: Legacy Pass Plugins

2. Present: Modern Pass Plugins

3. Future: From Passes to Extensions?
   - → Motivation
   - → Proposals

Round Tables



Tuesday 5 PM, right after the talk

Another one on Wednesday

Plugin term is convoluted

`ld.lld -plugin gold.so` ⛔ LTO plugin

`clang -fplugin=clad.so` ⛔ Clang frontend plugin

`clang -fpass-plugin=omvll.so` ✅ Pass-Plugin

Past

“ Don’t dwell in the past

Don’t dream of the future

Concentrate the mind on

the present moment. Focus!

404 @ Phabricator 😄

Past: Legacy Pass Plugins

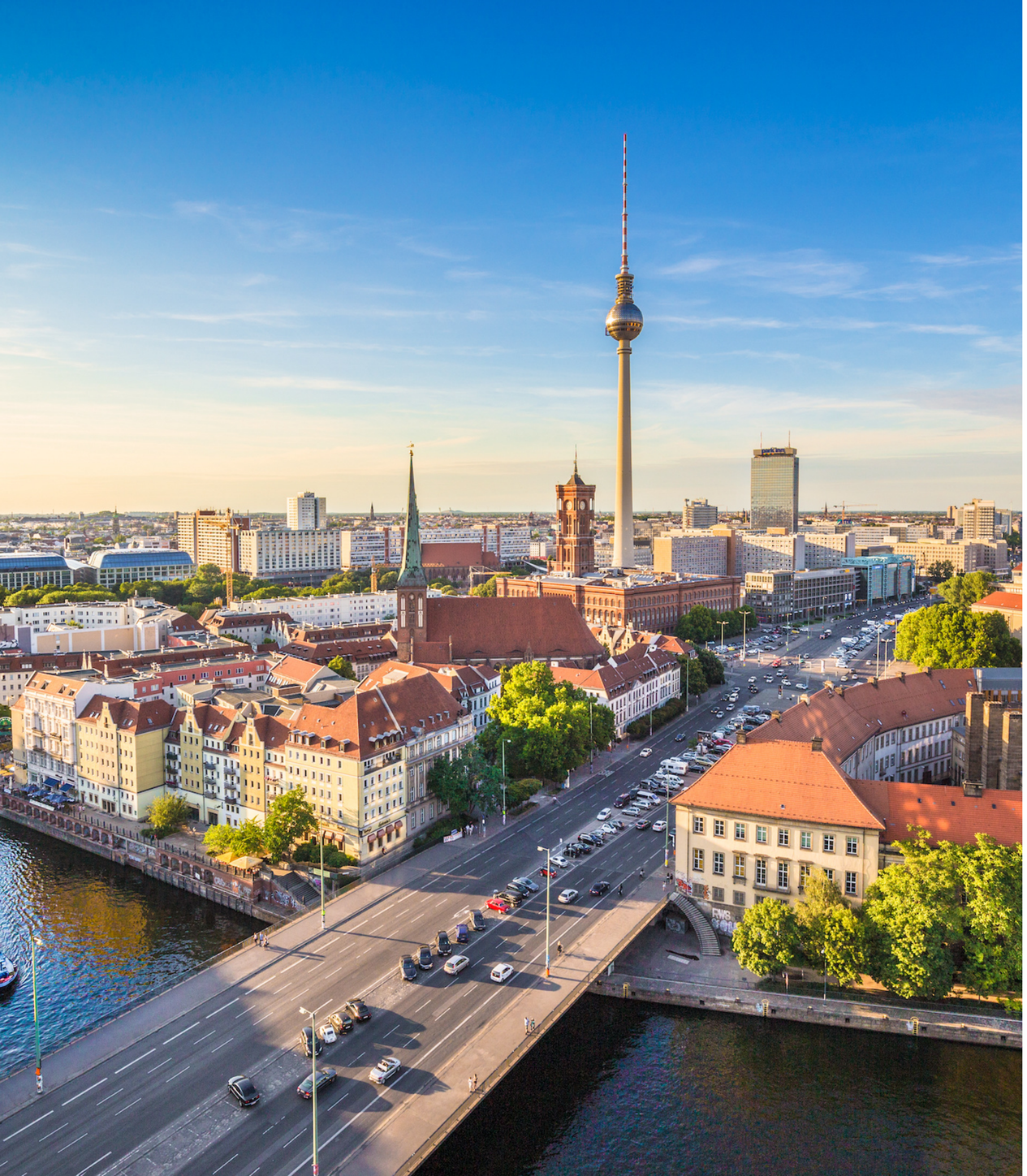
Option `-load` dates back to the early 2000s[1][2]

```
struct PluginLoader {
  void operator=(const std::string &Filename);
  static unsigned getNumPlugins();
  static std::string& getPlugin(unsigned num);
};

static cl::opt<PluginLoader, false, cl::parser<std::string>>
    LoadOpt("load", cl::value_desc("pluginfilename"),
            cl::desc("Load the specified plugin"));
```

Past: Legacy Pass Plugins

Plugins like Polly used static init to register new passes [1] [2]

```
class StaticInitializer {
public:
  StaticInitializer() {
    llvm::PassRegistry &Registry = *PassRegistry::getPassRegistry();
    polly::initializePollyPasses(Registry);
  }
};

static StaticInitializer InitializeEverything;
```



Present

Modern Pass Plugins

Plugin Renaissance with the New Pass Manager

Major contributions

2017 **Philip Pfaffe** adds a pass registration mechanism for Polly[1]

2018 He refines it into a plugin API so that "*interaction with a plugin is always initiated from the tools perspective*"[2]

2020 **Serge Guelton** generalizes it, removes remaining Polly-specific code from LLVM and adds an Bye 👋 example[3]

Modern Pass Plugins

On the plugin side, we implement a defined interface

```
extern "C" struct PassPluginLibraryInfo {
  uint32_t APIVersion;
  const char *PluginName;
  const char *PluginVersion;
  void (*RegisterPassBuilderCallbacks)(PassBuilder &);
};
```

Modern Pass Plugins

```
extern "C" PassPluginLibraryInfo llvmGetPassPluginInfo() {
    return {LLVM_PLUGIN_API_VERSION, "Bye", LLVM_VERSION_STRING,
            [](PassBuilder &PB) {
              PB.registerVectorizerStartEPCallback(
                  [](FunctionPassManager &PM, OptimizationLevel Level) {
                    PM.addPass(Bye());
                  });
              PB.registerPipelineParsingCallback(
                  [](StringRef Name, FunctionPassManager &PM, ...) {
                    if (Name == "goodbye") {
                      PM.addPass(Bye());
                      return true;
                    }
                    return false;
                  });
            }};
```

Modern Pass Plugins

On the tools side, we load them explicitly

```
static cl::list<std::string>
    PassPlugins("load-pass-plugin",
                cl::desc("Load passes from plugin library"));

for (auto &PluginFN : PassPlugins) {
  auto PassPlugin = PassPlugin::Load(PluginFN);
  if (!PassPlugin)
    continue;
  PassPlugin->registerPassBuilderCallbacks(PB);
}
```

Modern Pass Plugins: Pros

- Use `PassBuilder` the same way as in-tree tools
- Same concept in MLIR: entry-point `mlirGetPassPluginInfo()`
- Keep existing benefits:
  - C interface for plugin registration
  - Fast and easy builds against LLVM release versions

Modern Pass Plugins: Cons

Pass base class and `PassBuilder` definitions are C++

```
struct Bye : PassInfoMixin<Bye> {
  PreservedAnalyses run(Function &F, FunctionAnalysisManager &) {
    if (!runBye(F))
      return PreservedAnalyses::all();
    return PreservedAnalyses::none();
  }
};
```

Building Plugins correctly isn't trivial

Plugin binaries must fit target compiler's C++ ABI

Modern Pass Plugins: Tools

- `opt -load=/path/to/Bye.so -passes=goodbye`
  `opt -load-pass-plugin=/path/to/Bye.so`
  docs/CommandGuide/opt.html#cmdoption-opt-load
- `clang -fpass-plugin=/path/to/Bye.so`
  docs/ClangCommandLineReference.html#cmdoption-clang-fpass-plugin
- `flang -fpass-plugin=/path/to/Bye.so`
  docs/FlangCommandLineReference.html#cmdoption-flang-fpass-plugin
- `clang-repl -fpass-plugin=/path/to/Bye.so`
  weliveindetail.github.io/blog/post/2024/08/29/omvll-clang-repl.html

Modern Pass Plugins: Tools

- `swiftc -load-pass-plugin=/path/to/bye.so`
  mainline: swiftlang/swift/pull/68985

- `rustc -Zllvm-plugins=/path/to/bye.so`
  unstable: llvm.plugins = true option rust-lang.zulipchat.com

- `ld.lld --load-pass-plugin=/path/to/Bye.so` (since 15.x[1])
  undocumented 🤷

Modern Pass Plugins: in the wild

Open-source projects:

- https://github.com/llvm/llvm-project/tree/release/20.x/polly
- https://github.com/EnzymeAD/Enzyme
- https://github.com/open-obfuscator/o-mvll

Yes, it's a niche for sure. But it might also a be chicken-egg-problem..

Future

From Passes to

Rich out-of-tree Extensions?

Motivation

Claim: There is a demand for domain-specific compiler extensions.

Evidence? Looking at sanitizers:

```
2017 asan,dfsan,msan,tsan,safestack,cfi,esan,scudo
2018 asan,dfsan,msan,hwasan,tsan,safestack,cfi,esan,scudo,ubsan
2019 asan,dfsan,msan,hwasan,tsan,safestack,cfi,esan,scudo,ubsan
2020 asan,dfsan,msan,hwasan,tsan,safestack,cfi,scudo,ubsan,gwp_asan
2021 asan,dfsan,msan,hwasan,tsan,safestack,cfi,scudo,ubsan,gwp_asan
2022 asan,dfsan,msan,hwasan,tsan,safestack,cfi,scudo,ubsan,gwp_asan
2023 asan,dfsan,msan,hwasan,tsan,safestack,cfi,scudo,ubsan,gwp_asan
2024 asan,dfsan,msan,hwasan,tsan,safestack,cfi,scudo,ubsan,gwp_asan
2025 asan,rtsan,dfsan,msan,hwasan,tsan,tysan,safestack,cfi,scudo,ubsan,gwp_asan,nsan
          ^^^^^                          ^^^^^                                  ^^^^
```

Sanitizers: Who are the newcomers?

We can now check for:

- TypeSanitizer: type-based aliasing violations
- NumericalStabilitySanitizer: floating point precision issues
- RealtimeSanitizer: blocking calls in code with deterministic runtime

Observations: More domain-specific + less C/C++ specific

Future: Towards Rich Out-of-tree Extensions?

Should we build everything upstream forever?

Alternative: Could we implement extensions like Sanitizers as plugins?

```
[ ] Frontend: Attributes control where/how they apply (or not)
[x] IR Pass: Inject instrumentation, mostly calls into a runtime library
[ ] Driver: Add runtime library to the link line
```

Frontend with built-in Sanitizer (Realtime Sanitizer)

Clang: new `[[clang::(non)blocking]]` attributes translate to built-in `llvm::Attribute::SanitizeRealtime(Blocking)`
llvm-project/commit/f03cb005eb4b

Swift: `RTSanStandaloneSwift` package wraps C API in expression macros
swiftpackageindex.com/realtime-sanitizer/RTSanStandaloneSwift

Rust: `rtsan-standalone` crate wraps C API in procedural macros
crates.io/crates/rtsan-standalone

also preparing RFC for rustc built-in support
github.com/rust-lang/rfcs/pull/3766

Frontend without built-in Sanitizer

Clang:

- Frontend-Plugin could define attributes `[[clang::(non)blocking]]`
- Emit annotations instead of `llvm::Attribute::SanitizeRealtime`?
- Combine with Pass-Plugin in a single shared lib! github.com/vgvassilev/clad

Modern languages:

- Could language features emit annotations directly?

 **Round Table:** Can we use annotations?

Or could we teach Pass-Plugins to define LLVM attributes?

How to add a runtime library to the link line?

Named metadata entries for auto-linking might help:

- `llvm.linker.options`
  docs/LangRef.html#automatic-linker-flags-named-metadata
- `llvm.dependent-libraries`
  docs/LangRef.html#dependent-libs-named-metadata

 **Round Table:** Can we make it consistent or find a better way?

Future: Rich out-of-tree Extensions?

Doesn’t seem impossible!

```
[x] Frontend: Attributes control where/how they apply (or not)
[x] IR Pass: Inject instrumentation, mostly calls into a runtime library
[x] Driver: Add runtime library to the link line
```

Realtime Sanitizer: What is the story?

- Start a **hack in a fork** and reach a PoC
- Promote in domain-specific communities and find interested contributors
  adc23.sched.com/event/1PudD/radsan-a-realtime-safety-sanitizer
- Write RFC discourse.llvm.org/t/rfc-nolock-and-noalloc-attributes/76837/
- RFC considered to mature downstream or implement outside of LLVM[1]
- #92460 merged upstream in May 2024 and **150+ PRs** since

Realtime Sanitizer: Developer perspective

Pro upstream:

- **re-use infrastructure from other sanitizers**
- reviewers give guidance, help find issues and propose improvements
- boost reachable audience and get maximum convenience for users
- no immediate commercial interests (apparently)

Downside:

- requirements on code-quality and cross-platform support
- extra complexity from considering interference with other sanitizers

Future: Proposals

If we want to promote out-of-tree extensions, we could:

1. Provide re-usable infrastructure
2. Make it a playground to test new ideas
3. Motivate vendors to support plugins!

1. Re-usable infrastructure for out-of-tree extensions

Today:

- Plugin interface: unit-tests + Bye example with LIT tests
- Most bots don't build examples
- Most vendors don't ship examples
- Bye is quite primitive

Make it a **Reference Plugin**, that is built and deployed by default?

1. Re-usable infrastructure for out-of-tree extensions

Reference Plugin:

(1) Complexity of real-world extension

(2) Should work for LLVM and MLIR

(3) Do something useful for experimentation

(4) Consider a pure C interface?

2. Playground to test new ideas and not fork LLVM

Load a Python script and run it in as a pass?

(1) Python is popular + real-world complexity (e.g. static libPython?)

(2) Bindings for LLVM and MLIR

(3) Write IR transforms without building the plugin!

Two open-source repos with proof-of-concept:

- C++ with Numba's llvmlite: github.com/weliveindetail/llvm-py-pass
- Rust with llvmcpy from rev.ng: github.com/aneeshdurg/pyllvmpass

3. Motivate vendors to support and ship plugins!

Concerns:

- Security and tampering with internals (probably Apple)

  → Would code-signing checks help?

- Compatibility, versioning and dependence (probably Rust)

  → Would a pure C API version help?

 **Round Table:** Let's keep dreaming of a bright future for a bit!

Pass Plugins

Round Tables

Tuesday 5 PM, right after the talk

Another one on Wednesday

Contact

stefan.graenitz@gmail.com

weliveindetail.github.io/blog/about/