The llvm-autojit plugin

Compile your code at runtime. Automagically ✨

https://github.com/weliveindetail/llvm-autojit

Stefan Gränitz / LLVM Meetup Berlin / 20 November 2025

bzip2 is a good example

https://github.com/miyuki/spec-cpu2006-redist/tree/master/original/401.bzip2/bzip2-1.0.3

- Plain C, no dependencies, multiple compile-units
- Builds and links an archive
- Few different code paths, e.g. --help and --compress
- Input file size scales workload, result is verifiable:

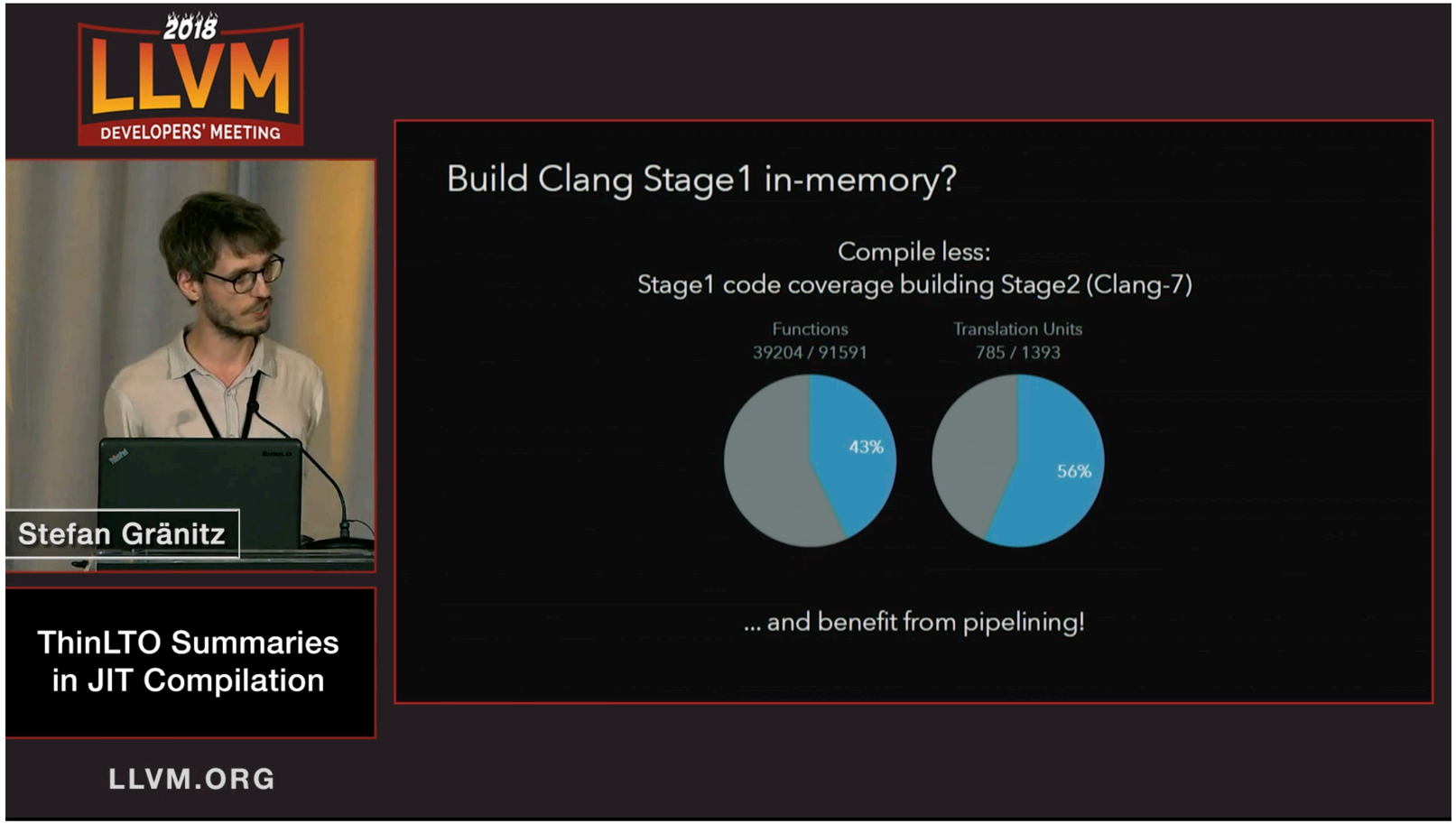
```
> bzip2 --compress test.txt
> md5sum --check test.md5
test.txt.bz2: OK
```

Original idea

Can we speed up one-shot builds with JIT compilation?

Original idea

Can we speed up one-shot builds with JIT compilation?



Original idea

Can we speed up one-shot builds with JIT compilation?

- Pass plugin that cuts out as much code as possible from each module
- Original IR code is stored on disk: /tmp/autotjit_2d1387ca92e7b.bc

llvm-autojit transform

Run plugin in opt

```
> AUTOJIT_DEBUG=On opt --load-pass-plugin=/usr/local/lib/autojit.so -passes=autojit -S bzip2/compress.ll
autojit-plugin: /tmp/autojit_4f0e030a358521399cbed216e6c35c26_incoming.ll (source: bzip2/compress.ll)
autojit-plugin: Processing module bzip2/compress.ll
autojit-plugin: Keep variable stderr
autojit-plugin: Keep variable .str
autojit-plugin: Keep variable .str.1
autojit-plugin: Keep variable .str.2
autojit-plugin: Keep variable .str.3
autojit-plugin: Keep variable .str.4
autojit-plugin: Keep variable .str.5
autojit-plugin: Keep variable .str.6
autojit-plugin: Keep variable .str.7
autojit-plugin: Keep variable .str.8
autojit-plugin: Keep variable .str.9
autojit-plugin: Keep variable .str.10
autojit-plugin: /tmp/autojit_4f0e030a358521399cbed216e6c35c26.ll (source: bzip2/compress.ll)
autojit-plugin: Lazify function BZ2_compressBlock as __autojit_fn_13853860483258788042
autojit-plugin: /tmp/autojit_4f0e030a358521399cbed216e6c35c26_static.ll (source: bzip2/compress.ll)
```

Original idea

Can we speed up one-shot builds with JIT compilation?

- Pass plugin that cuts out as much code as possible from each module
- Original IR code is stored on disk: /tmp/autotjit_2d1387ca92e7b.bc
- **Replace function bodies with lazy materialization requests**

llvm-autojit transform

Input

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local void @BZ2_compressBlock(ptr noundef %0, i8 noundef zeroext %1) #0 !dbg !180 {
  %3 = alloca ptr, align 8
  %4 = alloca i8, align 1
  store ptr %0, ptr %3, align 8
    #dbg_declare(ptr %3, !183, !DIExpression(), !184)
  store i8 %1, ptr %4, align 1
    #dbg_declare(ptr %4, !185, !DIExpression(), !186)
  %5 = load ptr, ptr %3, align 8, !dbg !187
  %6 = getelementptr inbounds nuw %struct.EState, ptr %5, i32 0, i32 17, !dbg !189
  %7 = load i32, ptr %6, align 4, !dbg !189
  %8 = icmp sgt i32 %7, 0, !dbg !190
  br i1 %8, label %9, label %63, !dbg !190
  ...

138:                                              ; preds = %136, %112
  ret void, !dbg !311
}
```

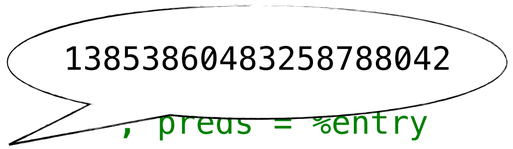
llvm-autojit transform

Output

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local void @BZ2_compressBlock(ptr noundef %0, i8 noundef zeroext %1) #0 {
entry:
  %existing_ptr = load ptr, ptr @__llvm_autojit_ptr_BZ2_compressBlock, align 8
  %2 = icmp eq ptr %existing_ptr, null
  br i1 %2, label %materialize, label %call

materialize:                                      ; preds = %entry
  store ptr inttoptr (i64 -4592883590450763574 to ptr), ptr @__llvm_autojit_ptr_BZ2_compressBlock, align 8
  call void @__llvm_autojit_materialize(ptr @__llvm_autojit_ptr_BZ2_compressBlock)
  %materialized_ptr = load ptr, ptr @__llvm_autojit_ptr_BZ2_compressBlock, align 8
  br label %call

call:                                             ; preds = %materialize, %entry
  %impl_ptr = phi ptr [ %existing_ptr, %entry ], [ %materialized_ptr, %materialize ]
  tail call void %impl_ptr(ptr %0, i8 %1)
  ret void
}
```



llvm-autojit transform

Output

```
@__llvm_autojit_ptr_BZ2_compressBlock = internal global ptr null
@__llvm_autojit_lazy_file = private unnamed_addr constant [49 x i8] c"/tmp/autojit_4f0e030a358521399cbed216e6c35c26.bc\00"

@llvm.global_ctors = appending global [1 x { i32, ptr, ptr }] [{ i32, ptr, ptr } {
  i32 100, ptr @_GLOBAL__sub_I_compress.c_llvm_autojit_init, ptr null
}]

; Function Attrs: nounwind
define internal void @_GLOBAL__sub_I_compress.c_llvm_autojit_init() #1 section ".text.startup" {
entry:
  call void @__llvm_autojit_register(ptr @__llvm_autojit_lazy_file)
  ret void
}

declare void @__llvm_autojit_materialize(ptr)
declare void @__llvm_autojit_register(ptr)
```

Original idea

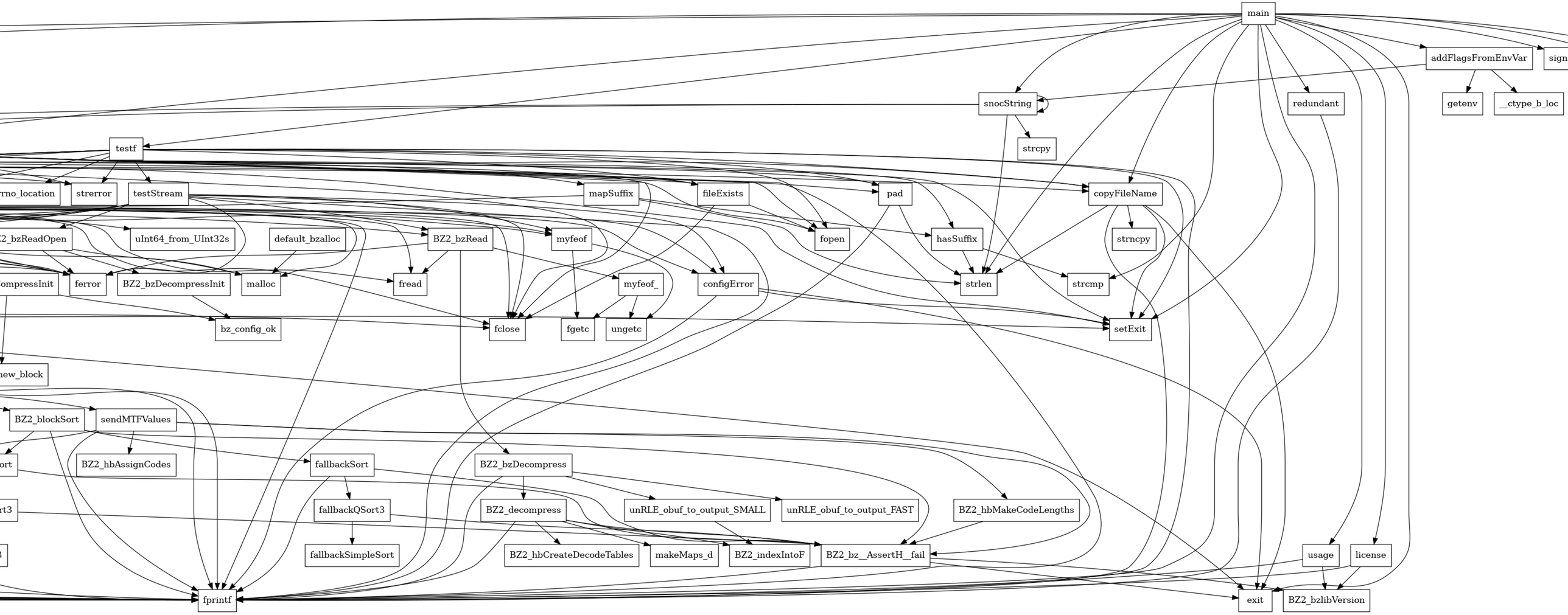
Can we speed up one-shot builds with JIT compilation?

- Pass plugin that cuts out as much code as possible from each module
- Original IR code is stored on disk: /tmp/autotjit_2d1387ca92e7b.bc
- Replace function bodies with lazy materialization requests
- **Remaining code is compiled statically and forms a "skeleton"**

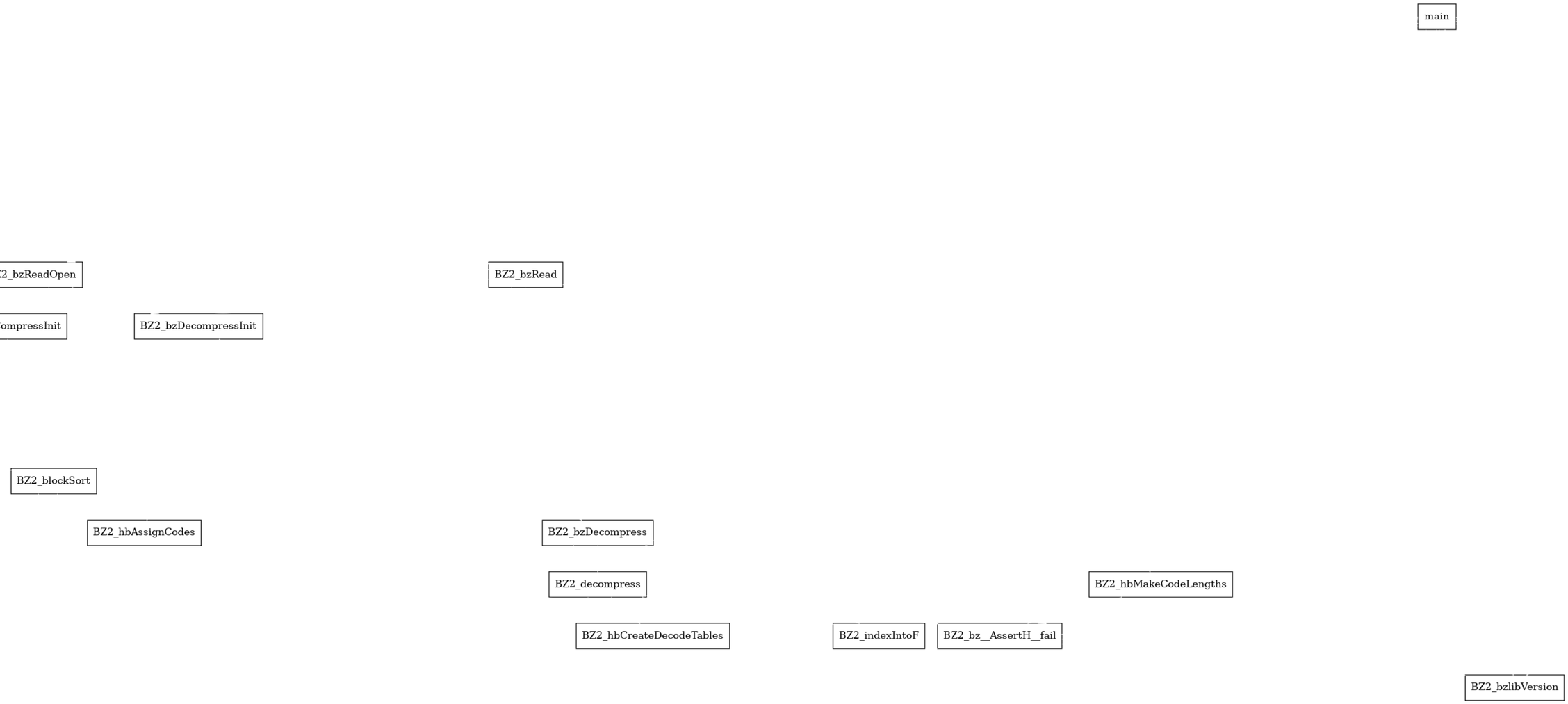
  ✓ Solves many problems like static initializers, link-once variables, breakpoints
- Original ABI remains, even on object-level (but will grow)

  ✓ autojit binaries can mix-and-match with regular binaries

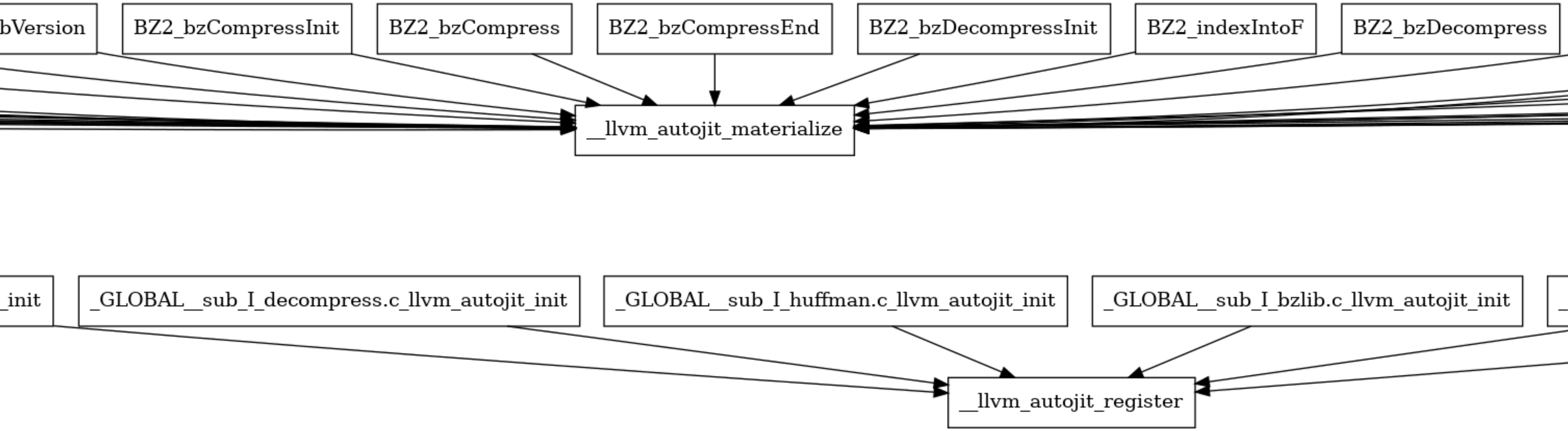
bzip2 callgraph



bzip2 callgraph



bzip2 callgraph



bzip2 binary size

```
$ ls -lh build_regular/bzip2
-rwxrwxr-x 1 ez ez 190K Nov 20 10:59 build_regular/bzip2

$ llvm-objdump -h build_regular/bzip2 > logs/bench-regular-sections.log
build_regular/bzip2:  file format elf64-x86-64

Sections:
Idx Name              Size     VMA              Type
  0                   00000000 0000000000000000
  1 .interp           0000001c 0000000000000318 DATA
  2 .note.gnu.property 00000020 0000000000000338
  3 .note.gnu.build-id 00000024 0000000000000358
  4 .note.ABI-tag     00000020 000000000000037c
  5 .gnu.hash         00000024 00000000000003a0
  6 .dynsym           00000468 00000000000003c8
  7 .dynstr           000001b1 0000000000000830
  8 .gnu.version      0000005e 00000000000009e2
  9 .gnu.version_r    00000050 0000000000000a40
 10 .rela.dyn         00000348 0000000000000a90
 11 .rela.plt         00000390 0000000000000dd8
 12 .init             0000001b 0000000000002000 TEXT
 13 .plt              00000270 0000000000002020 TEXT
 14 .plt.got          00000008 0000000000002290 TEXT
 15 .text             00014993 00000000000022a0 TEXT
 16 .fini             0000000d 0000000000016c34 TEXT
 17 .rodata           000024fa 0000000000017000 DATA
 18 .eh_frame_hdr     0000037c 00000000000194fc DATA
 19 .eh_frame         00000db0 0000000000019878 DATA
 20 .init_array       00000008 000000000001bdd0
 21 .fini_array       00000008 000000000001bdd8
 22 .dynamic          000001e0 000000000001bde0
 23 .got              00000040 000000000001bfc0 DATA
 24 .got.plt          00000148 000000000001c000 DATA
 25 .data             00000d10 000000000001c150 DATA
 26 .bss              00001130 000000000001ce60 BSS
 27 .comment          00000080 0000000000000000
 28 .debug_info       00004f55 0000000000000000 DEBUG
 29 .debug_abbrev     000009e8 0000000000000000 DEBUG
 30 .debug_line       00007e10 0000000000000000 DEBUG
 31 .debug_str        000014d7 0000000000000000 DEBUG
```

```
$ ls -lh build_autojit_shlib/bzip2
-rwxrwxr-x 1 ez ez 43K Nov 20 10:50 build_autojit_shlib/bzip2

$ llvm-objdump -h build_autojit_shlib/bzip2 > logs/bench-autojit-shlib-sections.log
build_autojit_shlib/bzip2:  file format elf64-x86-64

Sections:
Idx Name              Size     VMA              Type
  0                   00000000 0000000000000000
  1 .interp           0000001c 0000000000000318 DATA
  2 .note.gnu.property 00000020 0000000000000338
  3 .note.gnu.build-id 00000024 0000000000000358
  4 .note.ABI-tag     00000020 000000000000037c
  5 .gnu.hash         0000027c 00000000000003a0
  6 .dynsym           00000768 0000000000000620
  7 .dynstr           0000053d 0000000000000d88
  8 .gnu.version      0000009e 00000000000012c6
  9 .gnu.version_r    00000030 0000000000001368
 10 .rela.dyn         00000390 0000000000001398
 11 .rela.plt         00000030 0000000000001728
 12 .init             0000001b 0000000000002000 TEXT
 13 .plt              00000030 0000000000002020 TEXT
 14 .plt.got          00000008 0000000000002050 TEXT
 15 .text             00000fac 0000000000002060 TEXT
 16 .fini             0000000d 000000000000300c TEXT
 17 .rodata           00000201 0000000000004000 DATA
 18 .eh_frame_hdr     0000016c 0000000000004204 DATA
 19 .eh_frame         00000564 0000000000004370 DATA
 20 .init_array       00000038 0000000000005d98
 21 .fini_array       00000008 0000000000005dd0
 22 .dynamic          00000200 0000000000005dd8
 23 .got              00000028 0000000000005fd8 DATA
 24 .got.plt          00000028 0000000000006000 DATA
 25 .data             00000d10 0000000000006030 DATA
 26 .bss              00001248 0000000000006d40 BSS
 27 .comment          00000080 0000000000000000
 28 .debug_info       000012d7 0000000000000000 DEBUG
 29 .debug_abbrev     00000424 0000000000000000 DEBUG
 30 .debug_line       000003f8 0000000000000000 DEBUG
 31 .debug_str        0000065a 0000000000000000 DEBUG
```

bzip2 data section

```
Idx Name              Size     VMA              Type
  0                   00000000 0000000000000000
  1 .interp           0000001c 0000000000000318 DATA
  2 .note.gnu.property 00000020 0000000000000338
  3 .note.gnu.build-id 00000024 0000000000000358
  4 .note.ABI-tag     00000020 000000000000037c
  5 .gnu.hash         00000024 00000000000003a0
  6 .dynsym           00000468 00000000000003c8
  7 .dynstr           000001b1 0000000000000830
  8 .gnu.version      0000005e 00000000000009e2
  9 .gnu.version_r    00000050 0000000000000a40
 10 .rela.dyn         00000348 0000000000000a90
 11 .rela.plt         00000390 0000000000000dd8
 12 .init             0000001b 0000000000002000 TEXT
 13 .plt              00000270 0000000000002020 TEXT
 14 .plt.got          00000008 0000000000002290 TEXT
 15 .text             00014993 00000000000022a0 TEXT
 16 .fini             0000000d 0000000000016c34 TEXT
 17 .rodata           000024fa 0000000000017000 DATA
 18 .eh_frame_hdr     0000037c 00000000000194fc DATA
 19 .eh_frame         00000db0 0000000000019878 DATA
 20 .init_array       00000008 000000000001bdd0
 21 .fini_array       00000008 000000000001bdd8
 22 .dynamic          000001e0 000000000001bde0
 23 .got              00000040 000000000001bfc0 DATA
 24 .got.plt          00000148 000000000001c000 DATA
 25 .data             00000d10 000000000001c150 DATA
 26 .bss              00001130 000000000001ce60 BSS
 27 .comment          00000080 0000000000000000
 28 .debug_info       00004f55 0000000000000000 DEBUG
 29 .debug_abbrev     000009e8 0000000000000000 DEBUG
 30 .debug_line       00007e10 0000000000000000 DEBUG
 31 .debug_str        000014d7 0000000000000000 DEBUG
 32 .debug_addr       00000ca0 0000000000000000 DEBUG
 33 .debug_line_str   00000181 0000000000000000 DEBUG
 34 .debug_rnglists   0000001b 0000000000000000 DEBUG
 35 .debug_str_offsets 00001028 0000000000000000 DEBUG
 36 .symtab           00001470 0000000000000000
 37 .strtab           00000c2c 0000000000000000
 38 .shstrtab         0000018a 0000000000000000
```

```
Idx Name              Size     VMA              Type
  0                   00000000 0000000000000000
  1 .interp           0000001c 0000000000000318 DATA
  2 .note.gnu.property 00000020 0000000000000338
  3 .note.gnu.build-id 00000024 0000000000000358
  4 .note.ABI-tag     00000020 000000000000037c
  5 .gnu.hash         0000027c 00000000000003a0
  6 .dynsym           00000768 0000000000000620
  7 .dynstr           0000053d 0000000000000d88
  8 .gnu.version      0000009e 00000000000012c6
  9 .gnu.version_r    00000030 0000000000001368
 10 .rela.dyn         00000390 0000000000001398
 11 .rela.plt         00000030 0000000000001728
 12 .init             0000001b 0000000000002000 TEXT
 13 .plt              00000030 0000000000002020 TEXT
 14 .plt.got          00000008 0000000000002050 TEXT
 15 .text             00000fac 0000000000002060 TEXT
 16 .fini             0000000d 000000000000300c TEXT
 17 .rodata           00000201 0000000000004000 DATA
 18 .eh_frame_hdr     0000016c 0000000000004204 DATA
 19 .eh_frame         00000564 0000000000004370 DATA
 20 .init_array       00000038 0000000000005d98
 21 .fini_array       00000008 0000000000005dd0
 22 .dynamic          00000200 0000000000005dd8
 23 .got              00000028 0000000000005fd8 DATA
 24 .got.plt          00000028 0000000000006000 DATA
 25 .data             00000d10 0000000000006030 DATA
 26 .bss              00001248 0000000000006d40 BSS
 27 .comment          00000080 0000000000000000
 28 .debug_info       000012d7 0000000000000000 DEBUG
 29 .debug_abbrev     00000424 0000000000000000 DEBUG
 30 .debug_line       000003f8 0000000000000000 DEBUG
 31 .debug_str        0000065a 0000000000000000 DEBUG
 32 .debug_addr       00000198 0000000000000000 DEBUG
 33 .debug_line_str   0000017f 0000000000000000 DEBUG
 34 .debug_str_offsets 00000484 0000000000000000 DEBUG
 35 .symtab           00000de0 0000000000000000
 36 .strtab           000009fa 0000000000000000
 37 .shstrtab         0000017a 0000000000000000
```

Global variables remain static

✓ Solves extern data access and static initializers (in C++)

✓ Solves link-once symbols efficiently

```
> clang -O0 -g -fpass-plugin=/usr/local/lib/autojit.so -c bzlib.c
autojit-plugin: /tmp/autojit_a67b8472f07710730b1a65a68ac67bd7_incoming.ll (source: bzlib.c)
autojit-plugin: Processing module bzlib.c
autojit-plugin: Keep variable stderr
autojit-plugin: Keep variable .str
autojit-plugin: Keep variable .str.1
autojit-plugin: Keep variable .str.2
autojit-plugin: Keep variable .str.3
autojit-plugin: Keep variable .str.4
autojit-plugin: Keep variable .str.5
autojit-plugin: Keep variable stdin
autojit-plugin: Keep variable stdout
autojit-plugin: Promote variable bzerrorstrings as bzerrorstrings_autojit_module_a67b8472f07710730b1a65a68ac67bd7
autojit-plugin: Keep variable BZ2_crc32Table
autojit-plugin: Keep variable BZ2_rNums
autojit-plugin: Keep variable .str.6
autojit-plugin: Keep variable .str.7
...
```

Global variables remain static

✓ Solves extern data access and static initializers (in C++)

✓ Solves link-once symbols efficiently

```
> clang -O0 -g -fpass-plugin=/usr/local/lib/autojit.so -c bzlib.c
autojit-plugin: /tmp/autojit_a67b8472f07710730b1a65a68ac67bd7_incoming.ll (source: bzlib.c)
autojit-plugin: Processing module bzlib.c
autojit-plugin: Keep variable stderr
autojit-plugin: Keep variable .str
autojit-plugin: Keep variable .str.1
autojit-plugin: Keep variable .str.2
autojit-plugin: Keep variable .str.3
autojit-plugin: Keep variable .str.4
autojit-plugin: Keep variable .str.5
autojit-plugin: Keep variable stdin
autojit-plugin: Keep variable stdout
autojit-plugin: Promote variable bzerrorstrings as bzerrorstrings_autojit_module_a67b8472f07710730b1a65a68ac67bd7
autojit-plugin: Keep variable BZ2_crc32Table
autojit-plugin: Keep variable BZ2_rNums
autojit-plugin: Keep variable .str.6
autojit-plugin: Keep variable .str.7
...
```

Global variables remain static

Promote local definitions, so runtime can dlsym them

```
// Local definitions get exposed and must not collide. This affects both,
// static code and lazy code.
std::string UniquePostfix = "_autojit_module_" + getModuleGUID(M);
for (GlobalVariable &GV : M.globals()) {
  if (GV.hasAtLeastLocalUnnamedAddr())
    continue;
  // TODO: The assembler, emits 32-bit relocations for DSO-locals, which
  // will be out-of-range for our JITed code. The verifier prevents us from
  // removing the flag in some cases though.
  if (GV.hasDefaultVisibility() || GV.hasExternalWeakLinkage())
    GV.setDSOLocal(false);
  if (GV.hasLocalLinkage()) {
    GV.setName(GV.getName() + UniquePostfix);
    GV.setLinkage(GlobalValue::ExternalLinkage);
    continue;
  }
}
```

Global variables remain static

Promote local definitions, so runtime can dlsym them

```
// Local definitions get exposed and must not collide. This affects both,
// static code and lazy code.
std::string UniquePostfix = "_autojit_module_" + getModuleGUID(M);
for (GlobalVariable &GV : M.globals()) {
  if (GV.hasAtLeastLocalUnnamedAddr())
    continue;
  // TODO: The assembler, emits 32-bit relocations for DSO-locals, which
  // will be out-of-range for our JITed code. The verifier prevents us from
  // removing the flag in some cases though.
  if (GV.hasDefaultVisibility() || GV.hasExternalWeakLinkage())
    GV.setDSOLocal(false);
  if (GV.hasLocalLinkage()) {
    GV.setName(GV.getName() + UniquePostfix);
    GV.setLinkage(GlobalValue::ExternalLinkage);
    continue;
  }
}
```

2 processing phases of the plugin

```
> clang -O0 -g -fpass-plugin=/usr/local/lib/autojit.so -c bzlib.c
autojit-plugin: /tmp/autojit_a67b8472f07710730b1a65a68ac67bd7_incoming.ll (source: bzlib.c)
autojit-plugin: Promote variable bzerrorstrings as bzerrorstrings_autojit_module_a67b8472f07710730b1a65a68ac67bd7
autojit-plugin: Keep variable BZ2_crc32Table
autojit-plugin: Keep variable BZ2_rNums
autojit-plugin: Keep variable .str
autojit-plugin: Keep variable .str.1
autojit-plugin: Keep variable .str.2
...

autojit-plugin: /tmp/autojit_a67b8472f07710730b1a65a68ac67bd7.ll (source: bzlib.c)
autojit-plugin: Lazify function BZ2_bzlibVersion as __autojit_fn_4364279684691675085
autojit-plugin: Lazify function BZ2_bzCompressInit as __autojit_fn_2955341414002248116
autojit-plugin: Lazify function BZ2_bzCompress as __autojit_fn_18112044744406512933
autojit-plugin: Lazify function BZ2_bzCompressEnd as __autojit_fn_7492901234580991795
autojit-plugin: Lazify function BZ2_bzDecompressInit as __autojit_fn_8830962154303357271
...

autojit-plugin: /tmp/autojit_a67b8472f07710730b1a65a68ac67bd7_static.ll (source: bzlib.c)
```

Debugging features

Phase 1:

Common fixups for static and lazy code

Phase 2:

- Lazify static code
- Inject runtime support code

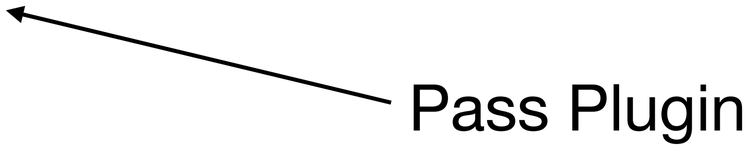
Phase 3: Dead-code elimination

```
extern "C" PassPluginLibraryInfo llvmGetPassPluginInfo() {
  return {.APIVersion = LLVM_PLUGIN_API_VERSION,
          .PluginName = "AutoJIT Pass",
          .PluginVersion = "v0.1",
          .RegisterPassBuilderCallbacks = [](PassBuilder &PB) {
            PB.registerPipelineStartEPCallback(
                [](ModulePassManager &MPM, OptimizationLevel Level) {
                  MPM.addPass(AutoJITPass());
                  MPM.addPass(GlobalDCEPass());
                });
          }};
}
```

Proof-of-concept release

https://github.com/weliveindetail/llvm-autojit/releases/tag/v0.1

```
> tar -C /usr/local -xzvf llvm-autojit-21_v0.1_x86-64.tar.gz
./bin/autojitd                                  } static out-of-process runtime
./lib/libautojit_static-x86_64.a
./lib/libautojit-runtime.so.21.1                } dynamic in-process runtime
./lib/libautojit-runtime.so
./lib/autojit.so
```



Pass Plugin

Basic Usage

https://github.com/weliveindetail/llvm-autojit#basic-cc-example

Basic C/C++ example

Create a `hello.c` with a simple hello world:

```
#include "stdio.h"

int main(void) {
  printf("Hello C\n");
  return 0;
}
```

Install a llvm-autojit release along with clang release 21 and compile a debug version with the following commands. They load the plugin `autojit.so` in the compiler (absolute path is required) and pass the runtime `libautojit-runtime.so` as a shared library to the linker. We need to add `/usr/local/lib` as shared library search path and specify `-rdynamic` for the runtime to resolve dynamic symbols from the executable.

```
> clang-21 -O0 -g -fpass-plugin=/usr/local/lib/autojit.so -o hello.o -c hello.c
> clang-21 -Wl,-rpath=/usr/local/lib -lautojit-runtime -rdynamic -o hello_c_shlib hello.o
./hello_c_shlib
[autojit-runtime] Loading module from cache /tmp/autojit_2045016cb90d1e65d71c2407a2570927.o (s
Hello C
```

Basic Rust example

LLVM plugins are not yet enabled in Rust release versions, but we can use a nightly version. The current plugin releases are tested with Rust nightly 1.91. Other versions may work as well. Please find more details in the Dockerfile for our CI image.

```
> rustc --version --verbose
rustc 1.91.0-nightly (1ebbd87a6 2025-08-11)
binary: rustc
commit-hash: 1ebbd87a62ce96a72b22da61b7c2c43893534842
commit-date: 2025-08-11
host: x86_64-unknown-linux-gnu
release: 1.91.0-nightly
LLVM version: 21.1.0
```

Once we have that, we create a `hello.rs` with a simple hello world:

```
fn main() {
  println!("Hello World!");
}
```

Compile and link in one step, then run the example:

```
> rustc -Z llvm-plugins="/usr/local/lib/autojit.so" \
        -L /usr/local/lib -lautojit-runtime
        -C link-arg=-Wl,-rpath,/usr/local/lib
        -C link-arg=-rdynamic
        -o hello_rust_shlib hello.rs
> ./hello_rust_shlib
[autojit-runtime] Loading module from cache /tmp/autojit_3c6eb49c3a3854d19574caf91dc7a72b.o (s
Hello World!
```

Benchmarks

https://github.com/weliveindetail/llvm-autojit#benchmark

```
> ./bzip2.sh --runs=10
Benchmark 401.bzip2 (shlib) in 10 runs
Compile-time regular:
  In 10 runs mean (min/max) time in seconds was: 0.944 (0.708 / 1.235)
Compile-time AutoJIT:
  In 10 runs mean (min/max) time in seconds was: 0.744 (0.537 / 1.018)

Binary sizes:
  Regular: 189 kb
  AutoJIT: 42 kb

Run-time regular:
  In 3 runs mean (min/max) time in seconds was: 2.611 (2.485 / 2.760)
323c74b2d7815a0b22979f45a93323e0  bzip2/outputs/data1_regular.txt.bz2
c3c5e912092b78a8a53faa34e9d7494e  bzip2/outputs/data2_regular.txt.bz2

Run-time AutoJIT:
  In 3 runs mean (min/max) time in seconds was: 2.745 (2.604 / 2.889)
323c74b2d7815a0b22979f45a93323e0  bzip2/outputs/data1_autojit.txt.bz2
c3c5e912092b78a8a53faa34e9d7494e  bzip2/outputs/data2_autojit.txt.bz2
```

Actual Usage

Example: https://github.com/nlohmann/json unit-tests

```
#/usr/bin/env bash

> CXX=clang++-21 cmake -G Ninja -S ../llvm-autojit-bench/json -B build \
                       -DCMAKE_BUILD_TYPE=Debug \
                       -DCMAKE_CXX_FLAGS="-fpass-plugin=/usr/local/lib/autojit.so" \
                       -DCMAKE_EXE_LINKER_FLAGS="-Wl,-rpath=/usr/local/lib -lautojit-runtime -rdynamic"

> ninja -C build -v tests/test-hash_cpp11

> ./build/tests/test-hash_cpp11 2>/dev/null
[doctest] doctest version is "2.4.12"
[doctest] run with "--help" for options
===============================================================================
[doctest] test cases: 2 | 2 passed | 0 failed | 0 skipped
[doctest] assertions: 2 | 2 passed | 0 failed |
[doctest] Status: SUCCESS!
```

Debugging

llvm-autojit runtime implements the GDB JIT Interface

```
> lldb -- build_autojit_shlib/bzip2
(lldb) target create "build_autojit_shlib/bzip2"
(lldb) b bzip2.c:1864
Breakpoint 1: no locations (pending).
WARNING:  Unable to resolve breakpoint to any actual locations.
(lldb) b main
Breakpoint 2: where = bzip2`main, address = 0x0000000000002210
(lldb) run --compress outputs/test_autojit.txt
Process 2527475 launched: 'build_autojit_shlib/bzip2' (x86_64)
Process 2527475 stopped
* thread #1, name = 'bzip2', stop reason = breakpoint 2.1
    frame #0: 0x0000555555556210 bzip2`main
bzip2`main:
->  0x555555556210 <+0>: push   rbp
    0x555555556211 <+1>: mov    rbp, rsp
    0x555555556214 <+4>: sub    rsp, 0x20
    0x555555556218 <+8>: mov    dword ptr [rbp - 0x14], edi
(lldb) bt
* thread #1, name = 'bzip2', stop reason = breakpoint 3.1
  * frame #0: 0x0000555555556210 bzip2`main
    frame #1: 0x00007ffff5829d90 libc.so.6`__libc_start_call_main(...) at libc_start_call_main.h:58:16
    frame #2: 0x00007ffff5829e40 libc.so.6`__libc_start_main_impl(...) at libc-start.c:392:3
    frame #3: 0x00005555555560e5 bzip2`_start + 37
```

Debugging

llvm-autojit runtime implements the GDB JIT Interface

```
> lldb -- build_autojit_shlib/bzip2
(lldb) target create "build_autojit_shlib/bzip2"
(lldb) b bzip2.c:1864
Breakpoint 1: no locations (pending).
WARNING:  Unable to resolve breakpoint to any actual locations.
...
(lldb) c
Process 2527475 resuming
1 location added to breakpoint 1
1 location added to breakpoint 2
Process 2527475 stopped
* thread #1, name = 'bzip2', stop reason = breakpoint 1.1 2.2
    frame #0: 0x00007ffff7f59087 JIT(0x7ffff57d4000)`main(argc=3, argv=0x00007fffffffd8f8) at bzip2.c:1864:28
   1861       configError();
   1862
   1863    /*-- Initialise --*/
-> 1864    outputHandleJustInCase  = NULL;
   1865    smallMode               = False;
   1866    keepInputFiles          = False;
   1867    forceOverwrite          = False;
(lldb) bt
* thread #1, name = 'bzip2', stop reason = breakpoint 1.1 2.2
  * frame #0: 0x00007ffff7f59087 JIT(0x7ffff57d4000)`main(argc=3, argv=0x00007fffffffd8f8) at bzip2.c:1864:28
    frame #1: 0x00007ffff5829d90 libc.so.6`__libc_start_call_main(...) at libc_start_call_main.h:58:16
    frame #2: 0x00007ffff5829e40 libc.so.6`__libc_start_main_impl(...) at libc-start.c:392:3
    frame #3: 0x00005555555560e5 bzip2`_start + 37
```

Feedback session

The archiver issue

llvm-autojit archives must be linked with --whole-archive (right now)

- linkers pull objects from archives on-demand, if they define any unresolved externals
- removing function bodies with llvm-autojit → drops unresolved externals → linker misses static objects
- Idea:
  - static constructors should force objects to be pulled
  - we use them for __autojit_register already
  - but it doesn't work yet.. 🤷

The thread-local storage issue

llvm-autojit skips modules with TLS right now (luckily they mix-and-match)

TLS is tricky:

- compiler emits TLS template for thread-local symbols
- dynamic loader runs `_dl_allocate_tls`, `_dl_allocate_tls_init`, etc. to setup a dedicated memory range for each thread
- access TLS variable implicitly via `__tls_get_addr`

The orc-runtime emulates it for JIT'd TLS variables:

llvm-project/llvm/lib/ExecutionEngine/Orc/ELFNixPlatform.cpp#L982

llvm-project/compiler-rt/lib/orc/elfnix_platform.cpp#L758

...but: ours are static and we want JIT'd code to use them!

How "much" ODR do we have to guarantee?

llvm-autojit cannot handle modules with TLS yet

Imagine this function in a header that is included in many compile-units:

```
inline int link_once_fn() {
  static int is_this_a_singleton_or_not = 0;
  return ++is_this_a_singleton_or_not;
}
```

Is it the compiler's responsibility to emit a single `static int` for the entire DSO?

Or is this UB? (i.e. user error)

Better solution for .hidden symbols

llvm-autojit currently promotes them, but that's not sustainable

Symbols with hidden visibility:

- are processed by the linker
- don't get exposed as dynamic symbols
- come in huge amounts (e.g. from the stdlib)

Promoting them to externals is heavy. So, discard the `dlsym` approach?

C++ has too many symbols

Crazy amounts of synthetic symbols slow down the plugin

nlohmann/json's test-hash_cpp11 sample:

- emits 3098 lazy functions
- of which 1679 functions get materialized

We barely reduce binary size:

```
> ls -lh build_regular/tests/test-hash_cpp11
-rwxrwxr-x 1 ez ez 3,3M Nov 20 16:49 build_regular/tests/test-hash_cpp11
> ls -lh build_autojit_shlib/tests/test-hash_cpp11
-rwxrwxr-x 1 ez ez 3,1M Nov 20 13:37 build_autojit_shlib/tests/test-hash_cpp11
```

The sanitizer issue

llvm-autojit breaks sanitizers

- llvm-autojit runs early in the pipeline to maximize compile-time profit
- sanitizer passes run later, so they only instrument remaining static code
- in fact, by default they run so late, that there is not even a pass-plugin entry-point behind them
- Ideas:
  1. Inject flag to run them earlier: `-sanitizer-early-opt-ep` (experimental)
  2. Run the sanitizer pass on lazy module on disk in isolation?

Can autojitd emit code to shared memory?

Sending all materialized code through a pipe is inefficient

Instead:

- Allocate range of shared memory between static stub and autojitd
- stub requests function body
- autojitd emits it in shared memory and returns address alone

Implement as "optimization" when stub and autojitd run on one machine?