Developers' Meeting

BERLIN 2025

Leveraging “nsw” in Flang’s LLVM IR Generation for Vectorization

Yusuke Minato

LLVM Developers’ Meeting – April 2025

FUJITSU

© 2025 Fujitsu Limited

Introduction

FUJITSU

- Vectorization for Flang
  - Flang: the Fortran frontend in the LLVM project
  - Flang relies on the LoopVectorize pass in the backend.
    - Its frontend itself doesn't have a vectorization pass.
  - We're trying to improve the capability of vectorization in the backend.
    - Vectorization plays an important role in optimizations.
- TSVC (Test Suite for Vectorizing Compilers)
  - Available in both Fortran and C
  - Helps distinguish frontend problems from backend issues
    - The frontend can affect vectorization because it may generate LLVM IR that is difficult for the backend to vectorize.

© 2025 Fujitsu Limited

Flang's Capability of Vectorization

FUJITSU

- Evaluation using TSVC
  - Options: `-O3 -ffast-math -march=armv9-a`
- Our contribution
  - Three additional loops can be vectorized since LLVM 20.
    - By introducing several options related to integer overflow into Flang

Vectorizable or not with		Number in LLVM 19	Number in LLVM 20
Clang	Flang		
Vectorizable	Vectorizable	76	80 (+4)
Vectorizable	**Non-Vectorizable**	**11**	**9 (-3+1)**
Non-Vectorizable	Vectorizable	1	1 (0)
Non-Vectorizable	Non-Vectorizable	40	38 (-2)

© 2025 Fujitsu Limited

Analysis of the Result

FUJITSU

- 11 loops fall into four categories:
  - A) **Suboptimal analysis for array subscripts (3 loops)**
  - B) Reduction variables passed by reference as real arguments (4 loops)
  - C) Gather/scatter with non-constant strides at compile-time (3 loops)
  - D) Loops requiring LTO only for Fortran version (1 loop)

- Categories A and B can be addressed in the frontend.
  - Adding more information to generated MLIR makes it easier for the LoopVectorize pass to vectorize the input LLVM IR.

© 2025 Fujitsu Limited

Address Calculations in Fortran

FUJITSU

- Array subscripts: 32 bits, Addresses: 1 word
  - 1 word is equal to 64 bits on 64-bit CPUs.
- Address calculations frequently involve different bit lengths.
  - Lower bounds of subscripts are rarely 0, unlike in C.
    - Address calculations need a step to calculate offsets from subscripts.
- Internal representation of array subscripts gets complicated.
  - This can significantly influence analyses and loop vectorization.

```
real a(n)
do i=1,n-1
  ... a(i+1) ...
end do
```



```
%16 = load i32, ptr %3, align 4 ;; i
%17 = add i32 %16, 1 ;; i + 1
%18 = sext i32 %17 to i64
%19 = sub nsw i64 %18, 1 ;; subtract the lower bound
%20 = mul nsw i64 %19, 1 ;; multiply by the stride
%21 = mul nsw i64 %20, 1 ;; multiply by the size of lower dims
%22 = add nsw i64 %21, 0
%23 = mul nsw i64 1, %7
%24 = getelementptr float, ptr %0, i64 %22 ;; a + ((i + 1) - 1L)
%25 = load float, ptr %24, align 4 ;; a(i+1)
```

↑ subscript

↓ address

Address calculations are linearized because the shape of an array is often mutable in Fortran.

© 2025 Fujitsu Limited

“nsw” in LLVM IR

FUJITSU

- Attribute on `add`/`sub`/`mul`/`shl`/`trunc` in LLVM IR
  - Shows the result will not overflow the range of signed integer
- Use case of `nsw`
  - Simplifying calculations involving different bit lengths
    - `(i + 1) - 1L == i` ?

- Do not add `nsw` to instructions whose results could overflow.
  - cf. Rust (release mode)
    - The behavior of integer overflow is defined (wraparound).

© 2025 Fujitsu Limited

“nsw” in LLVM IR

FUJITSU

- Attribute on `add`/`sub`/`mul`/`shl`/`trunc` in LLVM IR
  - Shows the result will not overflow the range of signed integer
- Use case of `nsw`
  - Simplifying calculations involving different bit lengths
    - `(i + 1) - 1L == i` ?
      - Where `i` = `INT_MAX`, (LHS) = `(long)INT_MIN - 1L`, (RHS) = `(long)INT_MAX`
      - Equivalent only if the addition has an `nsw` flag (i.e., `i <= INT_MAX - 1`)

- Do not add `nsw` to instructions whose results could overflow.
  - cf. Rust (release mode)
    - The behavior of integer overflow is defined (wraparound).

© 2025 Fujitsu Limited

Integer Overflow in Fortran

FUJITSU

- The Fortran standard does not mention integer overflow.
  - Leaves the handling of integer overflow up to compiler developers

- We can assume that some operations will never overflow:
  - Increments of DO loop variables
  - Calculations for array subscripts
  - Calculations for loop bounds
- However, code that violates our assumption may exist.
  - The option `-fwrapv` has been introduced into Flang, similar to Clang.

© 2025 Fujitsu Limited

Implementation Details

FUJITSU

- Introduced a new flag in `FirOpBuilder`, a kind of IRBuilder
  - The Builder lowers the AST to IR recursively.
  - This flag controls whether `nsw` is added to the target operations.
    - Caution: Fortran 2008 and later have intrinsics for bitwise comparisons.
- Patches
  - #91579, #110060, #113854, #110061, #118933

```
1899  1904     template <typename T>
1900  1905     hlfir::Entity
1901  1906     HlfirDesignatorBuilder::genSubscript(const Fortran::evaluate::Expr<T> &expr) {
      1907  +    fir::FirOpBuilder &builder = getBuilder();
      1908  +    mlir::arith::IntegerOverflowFlags iofBackup{};
      1909  +    if (!getConverter().getLoweringOptions().getIntegerWrapAround()) {
      1910  +      iofBackup = builder.getIntegerOverflowFlags();
      1911  +      builder.setIntegerOverflowFlags(mlir::arith::IntegerOverflowFlags::nsw);
      1912  +    }
1902  1913     auto loweredExpr =
1903  1914         HlfirBuilder(getLoc(), getConverter(), getSymMap(), getStmtCtx())
1904  1915             .gen(expr);
1905        -    fir::FirOpBuilder &builder = getBuilder();
      1916  +    if (!getConverter().getLoweringOptions().getIntegerWrapAround())
      1917  +      builder.setIntegerOverflowFlags(iofBackup);
```

```
779  +    auto iofi =
780  +        mlir::dyn_cast<mlir::arith::ArithIntegerOverflowFlagsInterface>(*op);
781  +    if (iofi) {
782  +      llvm::StringRef arithIOFAttrName = iofi.getIntegerOverflowAttrName();
783  +      if (integerOverflowFlags != mlir::arith::IntegerOverflowFlags::none)
784  +        op->setAttr(arithIOFAttrName,
785  +                    mlir::arith::IntegerOverflowFlagsAttr::get(
786  +                        op->getContext(), integerOverflowFlags));
787  +    }
```


© 2025 Fujitsu Limited

LLVM IR Example with nsw Required

FUJITSU

```
1  subroutine sample(a,lb,ub)
2    integer lb,ub,i
3    integer a(lb:ub)
4    do i=lb,ub-1
5      a(i+1) = i-lb
6    end do
7  end subroutine
```

```
1  define void @sample_(ptr captures(none) %0, ptr captures(none) %1, ptr captures(none) %2) {
2    %4 = load i32, ptr %1, align 4, !tbaa !4
3    %5 = sext i32 %4 to i64
4    %6 = load i32, ptr %2, align 4, !tbaa !10
5    %7 = sext i32 %6 to i64
6    %8 = sub i64 %7, %5
7    %9 = add i64 %8, 1
8    %10 = icmp sgt i64 %9, 0
9    %11 = select i1 %10, i64 %9, i64 0
10   %12 = sub nsw i32 %6, 1 ;; the upper bound of the loop
11   %13 = sext i32 %12 to i64
12   %14 = trunc i64 %5 to i32
13   %15 = sub i64 %13, %5
14   %16 = add i64 %15, 1
15   br label %17
16
17:                                               ; preds = %21, %3
18   %18 = phi i32 [ %32, %21 ], [ %14, %3 ]
19   %19 = phi i64 [ %33, %21 ], [ %16, %3 ]
20   %20 = icmp sgt i64 %19, 0
21   br i1 %20, label %21, label %34
```

```
22
23 21:                                               ; preds = %17
24   %22 = load i32, ptr %1, align 4, !tbaa !4
25   %23 = sub i32 %18, %22
26   %24 = add nsw i32 %18, 1 ;; the array subscript
27   %25 = sext i32 %24 to i64
28   %26 = sub nsw i64 %25, %5
29   %27 = mul nsw i64 %26, 1
30   %28 = mul nsw i64 %27, 1
31   %29 = add nsw i64 %28, 0
32   %30 = mul nsw i64 1, %11
33   %31 = getelementptr i32, ptr %0, i64 %29
34   store i32 %23, ptr %31, align 4, !tbaa !12
35   %32 = add nsw i32 %18, 1 ;; the increment of the DO variable
36   %33 = sub i64 %19, 1
37   br label %17
38
39 34:                                               ; preds = %17
40   ret void
41 }
```


© 2025 Fujitsu Limited

Challenges

FUJITSU

- Performance regression in the LoopStrengthReduce pass
  - While `sdiv` is changed to `udiv` in the InstCombine pass when considering `nsw` on its operands, it blocks analysis for the optimization.
    - https://github.com/llvm/llvm-project/issues/117318
- Remaining issues identified by TSVC (categories B and C)
  - Adding `nocapture` attribute to arguments
    - https://github.com/llvm/llvm-project/issues/106682
  - Accepting non-constant strides if they are found to be loop-invariant
    - https://github.com/llvm/llvm-project/issues/110611

© 2025 Fujitsu Limited

Acknowledgements

- This presentation is based on results obtained from a project, JPNP21029, subsidized by the New Energy and Industrial Technology Development Organization (NEDO).

© 2025 Fujitsu Limited

Thank you

© 2025 Fujitsu Limited

Developers' Meeting

BERLIN 2025

mobileye™

Instruction Cache Prefetching

Oriel Avraham

Compiler Engineer @ Mobileye

Agenda

- Instruction Cache
- Prefetch Instruction
- LLVM Analyses
- Performance Impact

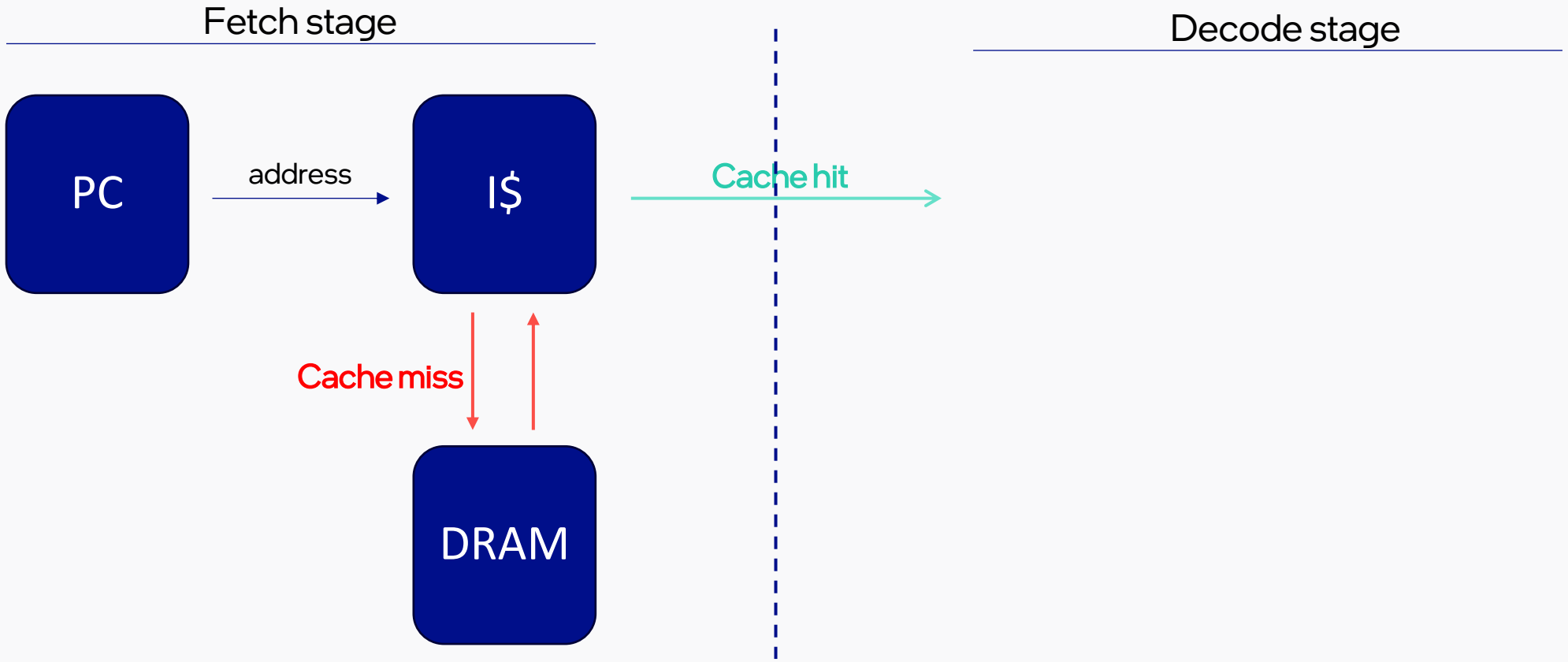
© MOBILEYE

1 Instruction Cache

© MOBILEYE

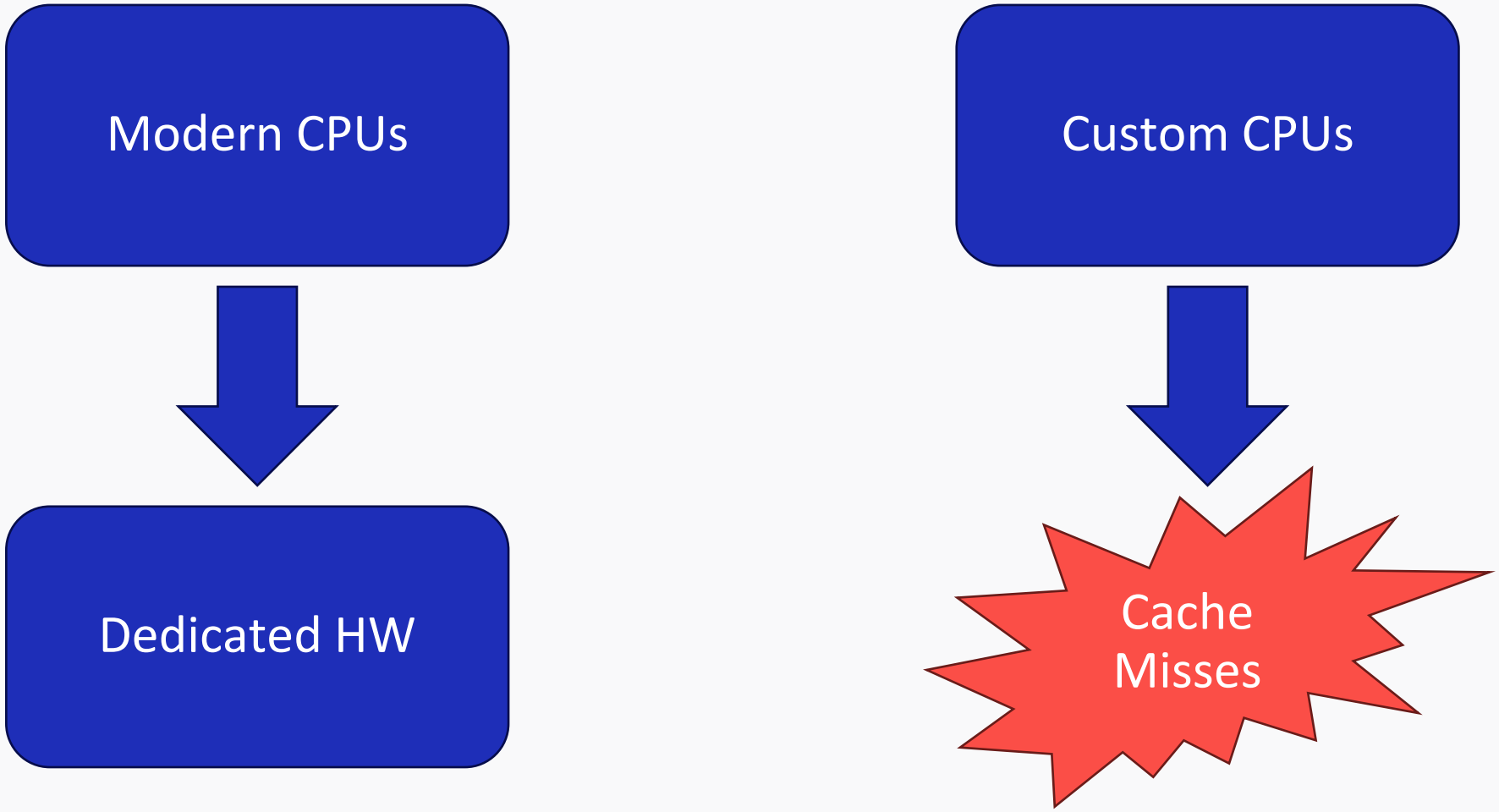
What is an Instruction Cache (I$)

A small, fast memory inside the CPU that stores recently fetched instructions.



© MOBILEYE

How do we keep the I$ Updated?



© MOBILEYE

2

Prefetch Instruction

© MOBILEYE

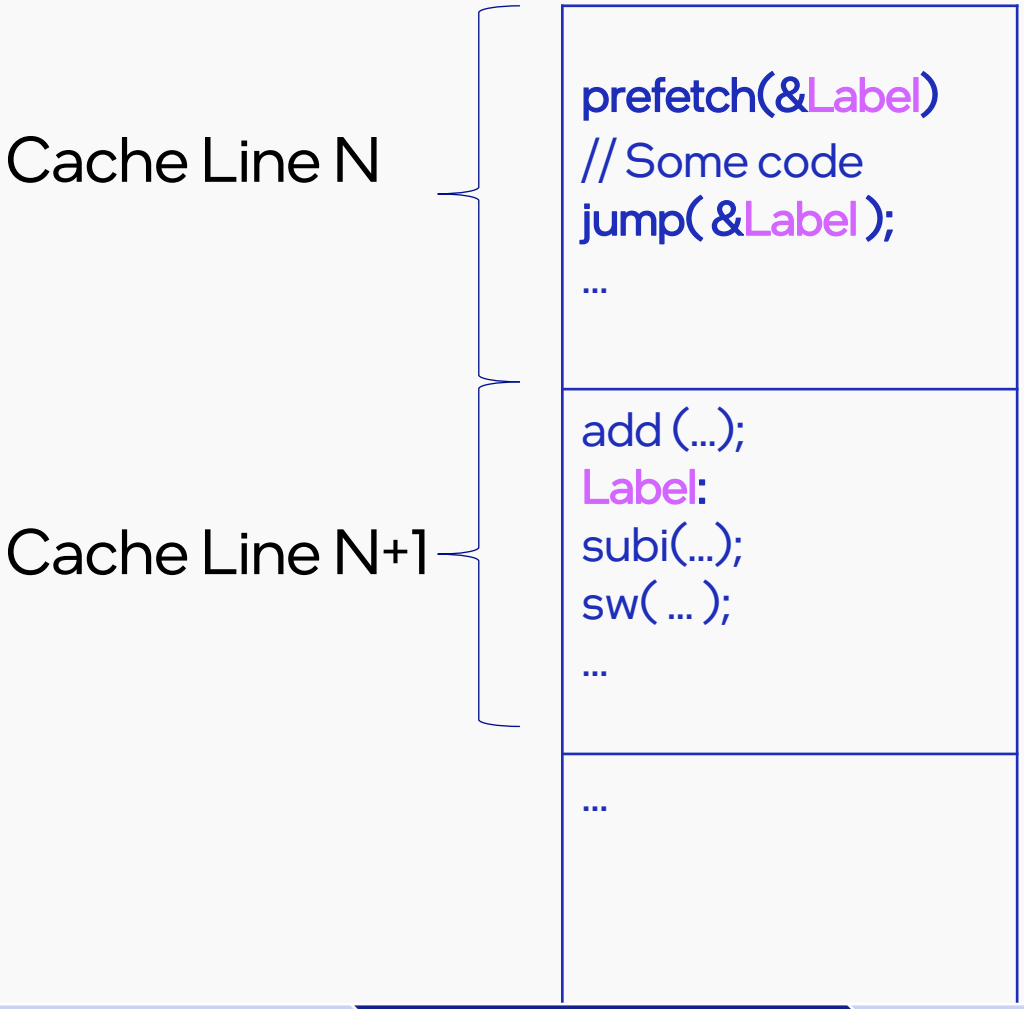
Prefetch Instruction - Specifications

A **prefetch instruction** is a **hint** to the processor to fetch instructions into the I$.

Syntax	prefetch(&Label)
Semantics	Signals to fetch the cache line associated with Label
Sync/Async	Asynchronous non-blocking behavior

© MOBILEYE

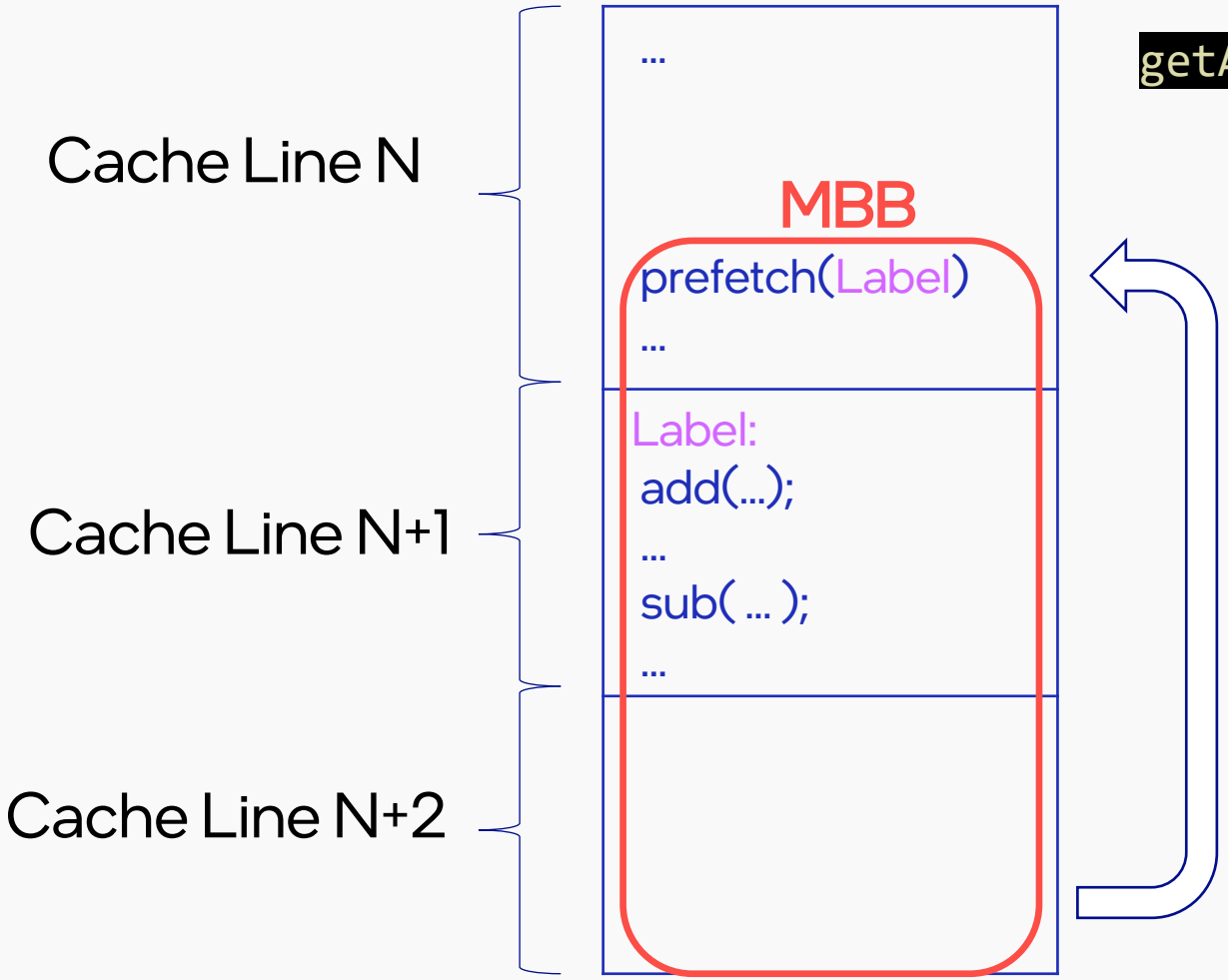
Example


Cache Line N
Cache Line N+1
prefetch(&Label)
// Some code
jump(&Label);
...
add (...);
Label:
subi(...);
sw(...);
...
...

LLVM Analyses

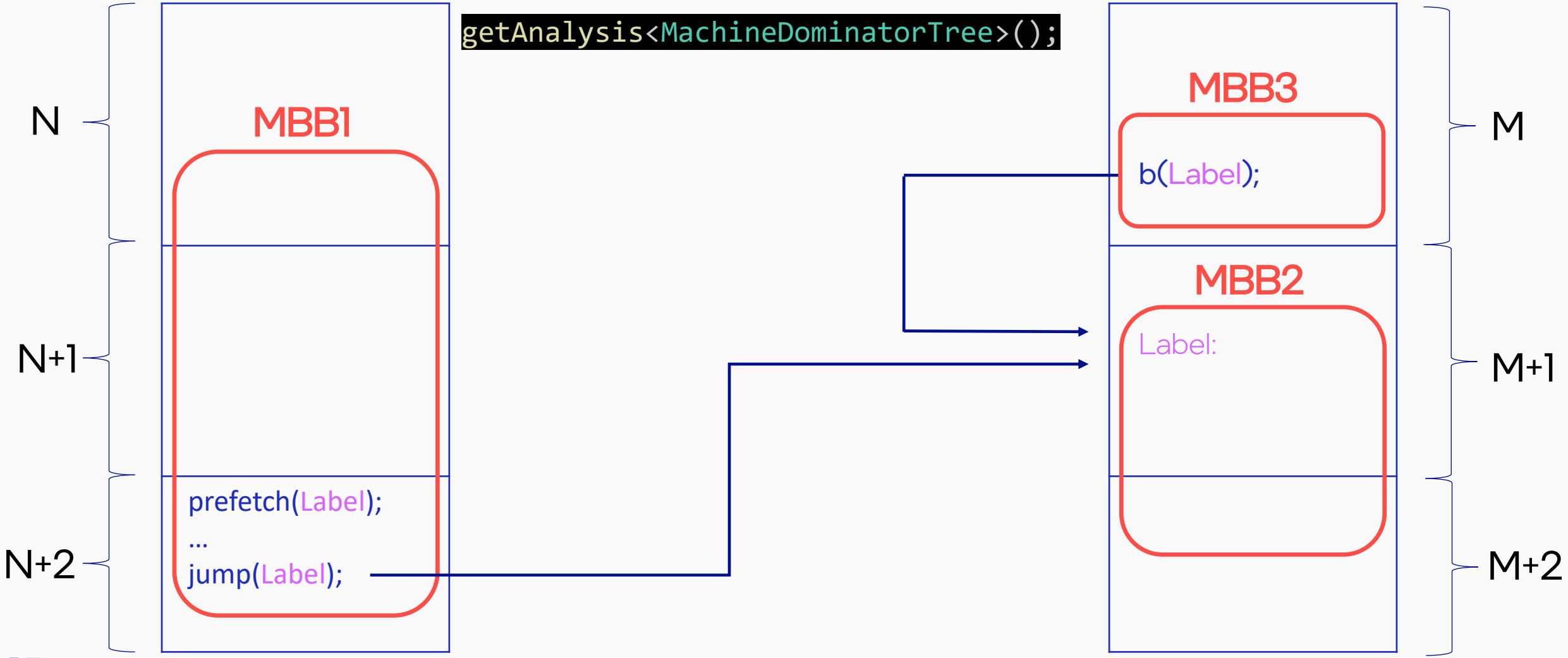
© MOBILEYE

Useful Analyses

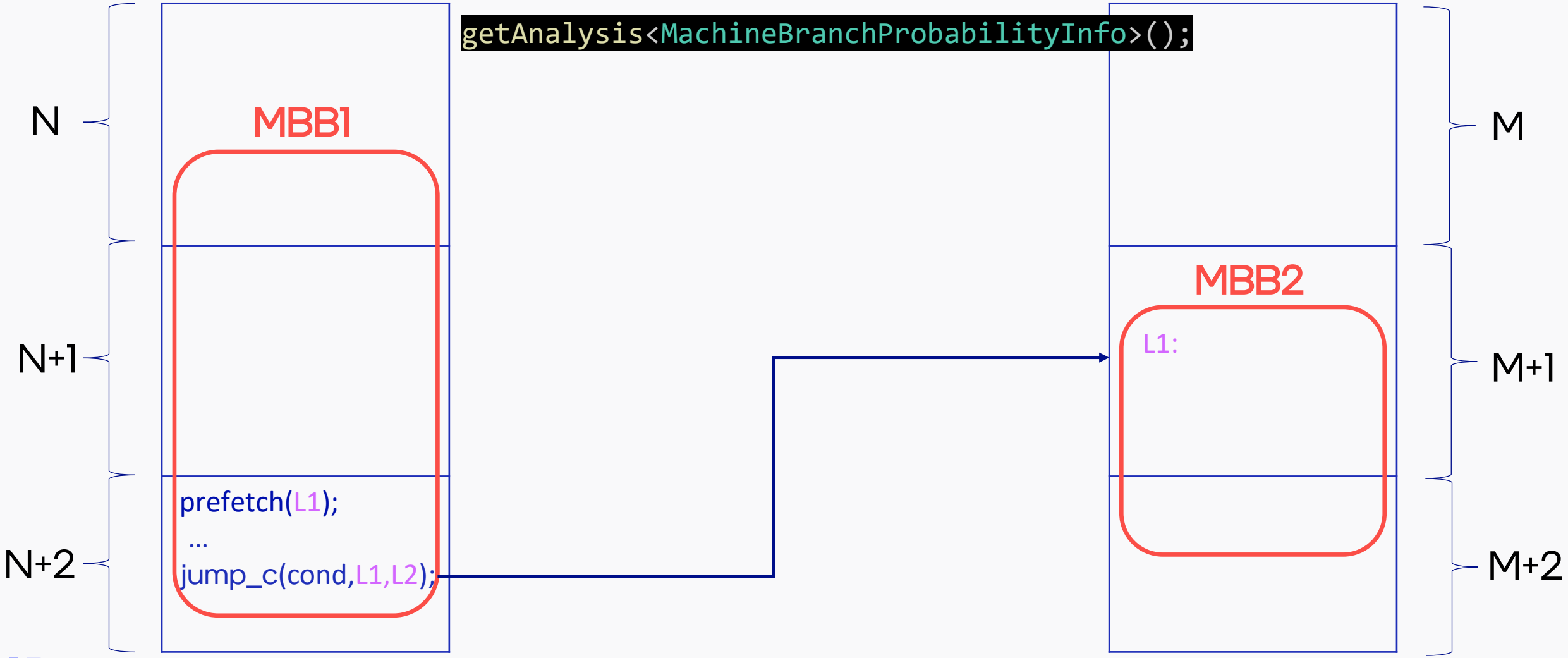



```
getAnalysis<MachineLoopInfo>();
```

Useful Analyses



Useful Analyses



4 Performance

© MOBILEYE

Performance Impact

I$ Performance

45% reduction in stalls

Total Performance

3-4%

© MOBILEYE

Conclusion

- Advanced architectures handle I$ efficiently, but simpler ones don’t.
- LLVM framework makes this optimization comfortable to implement.
- Compiler-driven prefetching is a viable software solution.

© MOBILEYE

mobileye™

Thank you!

oriel.avraham@mobileye.com

Developers' Meeting

BERLIN 2025

#embed in clang: one directive to embed them all

Mariya Podchishchaeva

@Fznamznon

What is #embed?

```
# embed <file-name>|”file-name” parameters...
```

parameters refers to the syntax of

no_arg/with_arg(values,...)/vendor::no_arg/vendor::with_arg(tokens...)

There are language-defined parameters, for example:

```
const int data[] = {
#embed "/dev/urandom" limit(512) // no more than 512 bytes
};
```

P.S. clang doesn’t support device files properly yet.

How is that supposed to work?

Users do:

```
const unsigned char data[] = {
#embed “data.bin“
};
```

The directive is expanded to comma-separated integer literals:

```
const unsigned char data[] = {
1, 2, 3
};
```

where 1, 2, and 3 are byte values from the resource.

How is that supposed to work?

Users do:

```
const unsigned char data[] = {
#embed “data.bin“
};
```

The directive is expanded to comma-separated integer literals:

```
const unsigned char data[] = {
1, 2, 3
};
```

where 1, 2, and 3 are byte values from the resource.

What is a ~~bug~~ big deal?

The answer is simple – this is very slow.

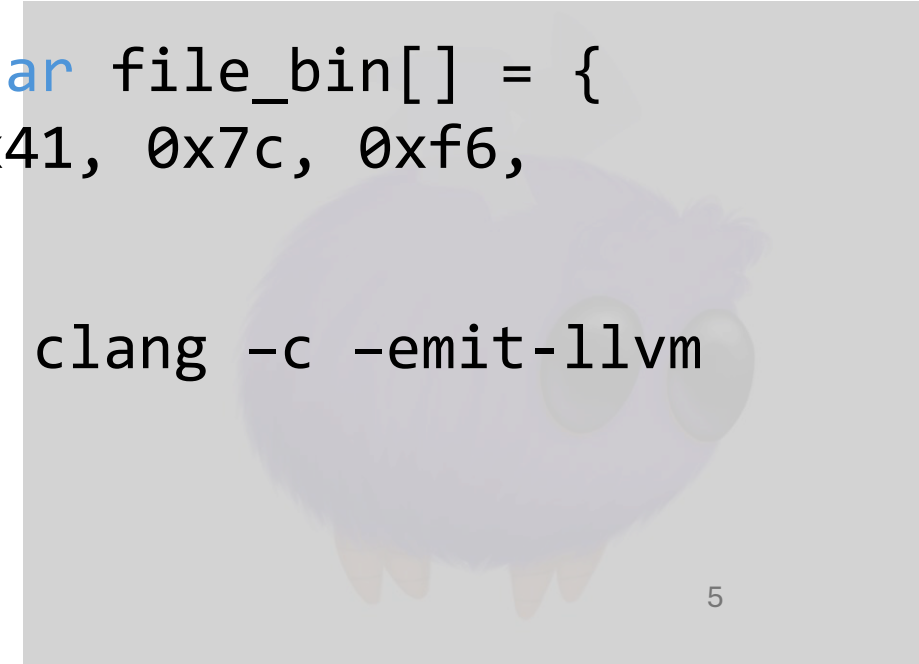
Let's do some comparison with "classic" methods…

```
head -c $((1024*1024*NUM_OF_MB)) /dev/urandom > file.bin
xxd -i file.bin > filexxd.c
```

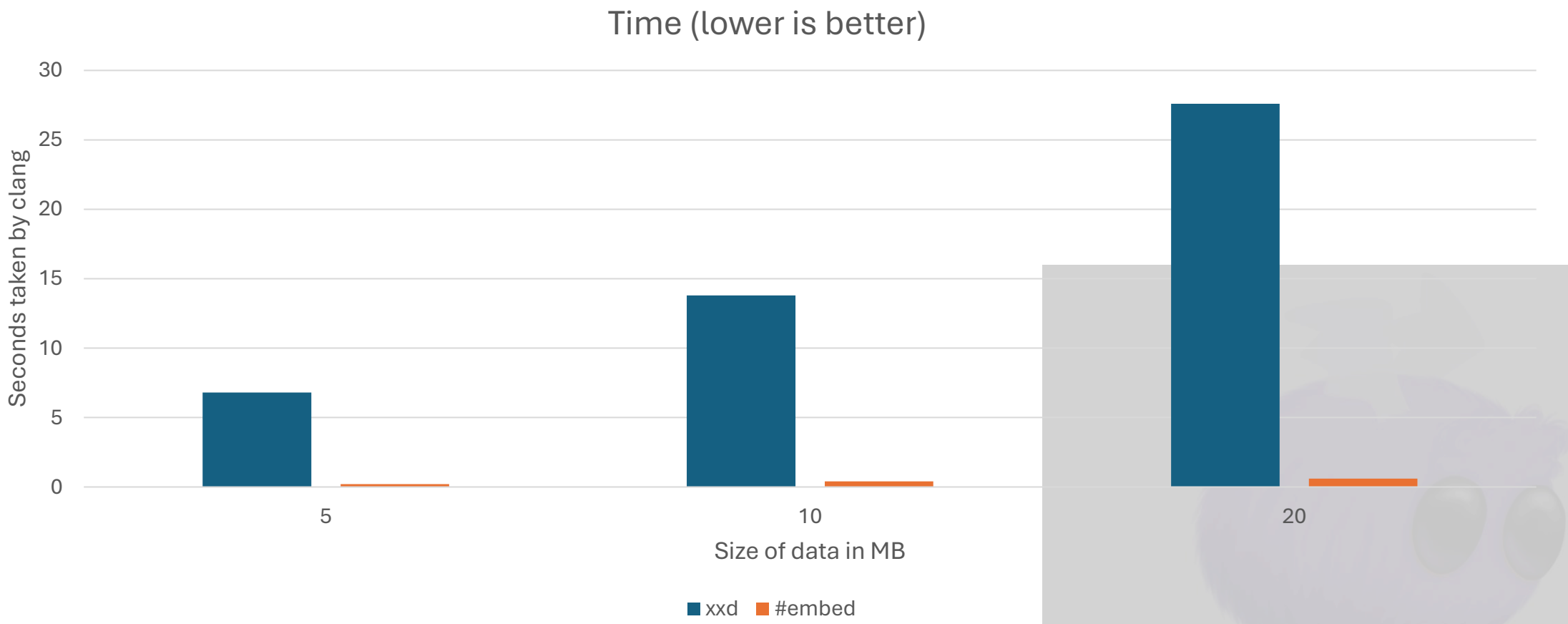
```
embed.c
unsigned char c[] = {
#embed "file.bin"
};
```

```
filexxd.c
unsigned char file_bin[] = {
   0x82, 0x41, 0x7c, 0xf6,
0x7c,…
```

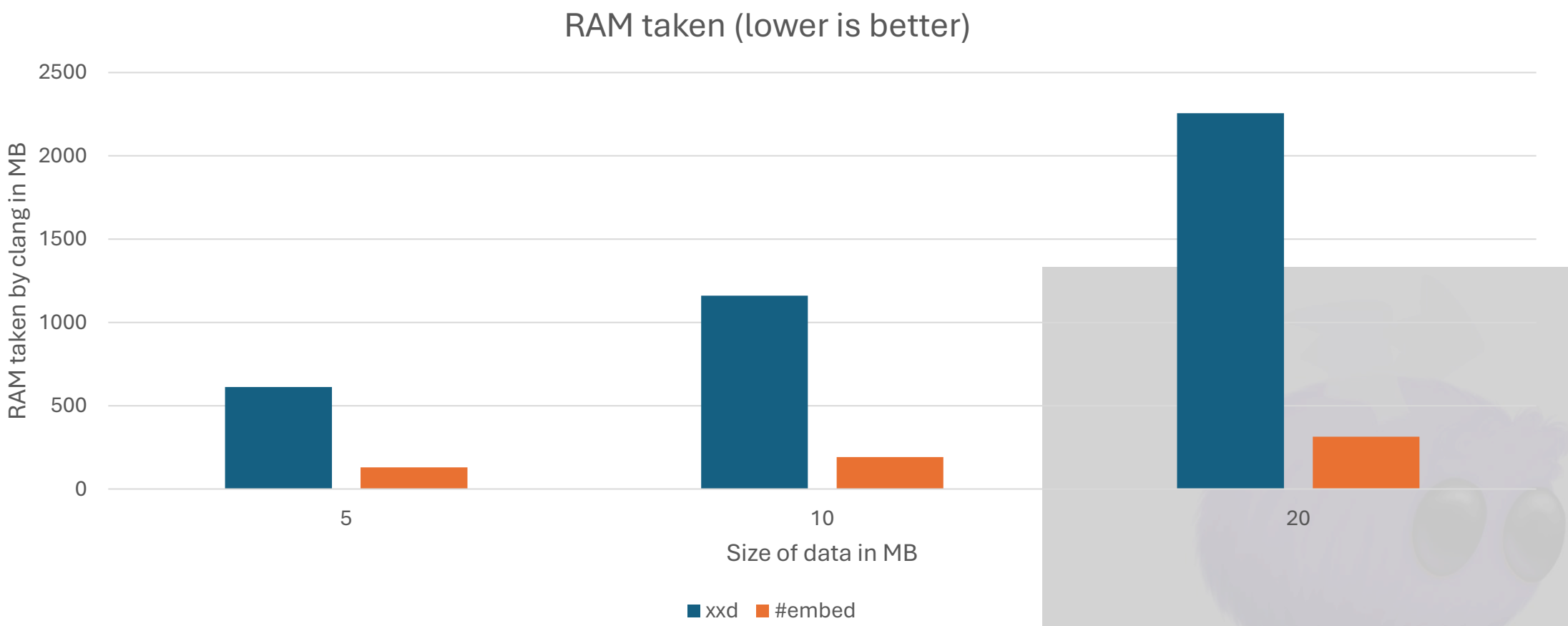
And compare `clang -c -emit-llvm embed.c` vs `clang -c -emit-llvm filexxd.c`



Time difference



RAM consumption difference



How did we get there?

```
unsigned char b[] = {
#embed __FILE__
};
```

```
`-VarDecl <line:1:1, line:3:1> line:1:15 b
'unsigned char[46]' cinit
  `-InitListExpr <col:21, line:3:1> 'unsigned
char[46]'
    `-StringLiteral <line:2:5> 'unsigned
char[46]' "unsigned char b[] = {\n    #embed
__FILE__\n};\n"
```

What to do when strings don’t work?

```
int a[2][3] = { 300,
#embed __FILE__
};
```

```
-VarDecl <line:2:1, line:4:1> line:2:5 a 'int[2][3]'
cinit
 `-InitListExpr <col:15, line:4:1> 'int[2][3]'
   |-InitListExpr <line:3:5> 'int[3]'
   | |-array_filler: ImplicitValueInitExpr 0x334a7360
'int'
   | `-EmbedExpr <col:5> 'int'
   |   |-begin: 0
   |   `-number of elements: 3
   `-InitListExpr <col:5> 'int[3]'
     |-array_filler: ImplicitValueInitExpr 0x334a7370
'int'
     `-EmbedExpr <col:5> 'int'
       |-begin: 3
       `-number of elements: 3
```

What is EmbedExpr?

- A reference to embedded data.
- Knows where to take the data and how many of it.
- Represents multiple bytes of data with a single expression.
- One InitListExpr may have several EmbedExprs referencing the same array of data but different parts of this array.
- Created only inside of InitListExpr.
- Handled by AST consumers similarly to array filler.

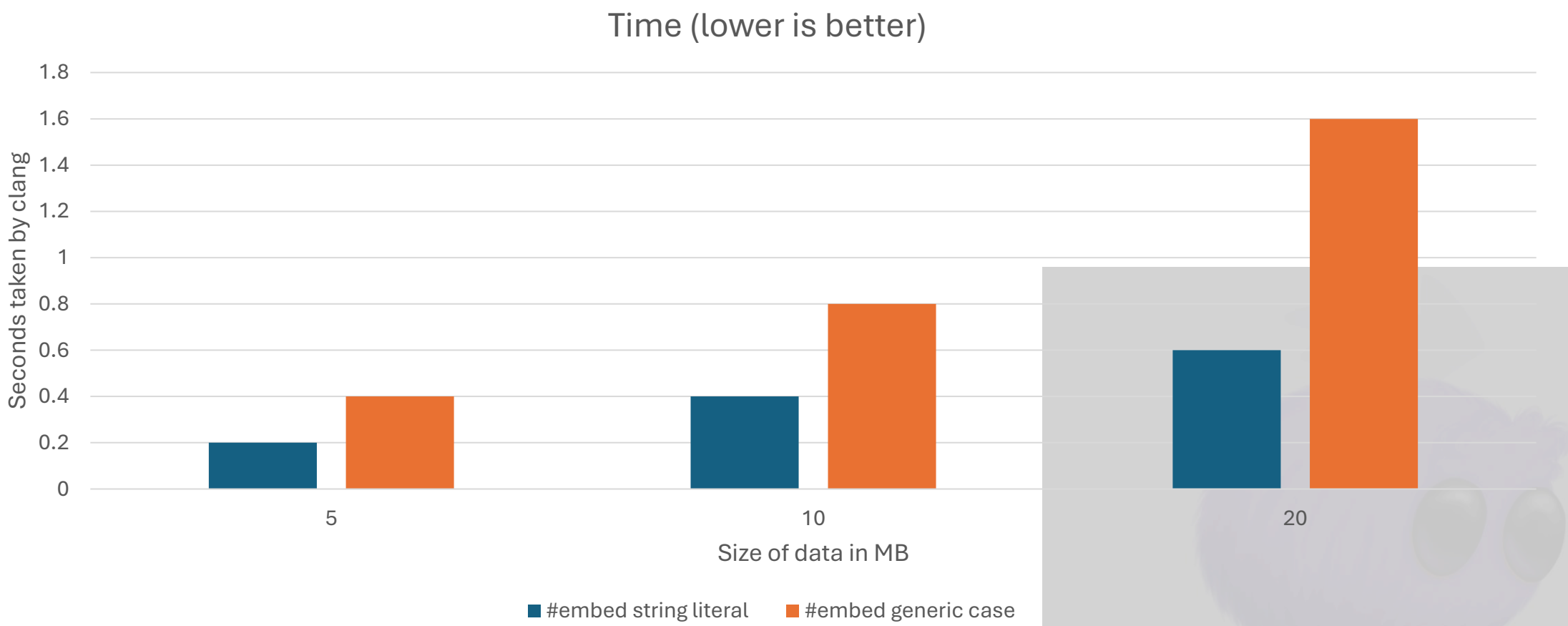
How expensive is that?

Let’s check how much time and RAM clang will take with EmbedExpr and compare it to StringLiteral case.

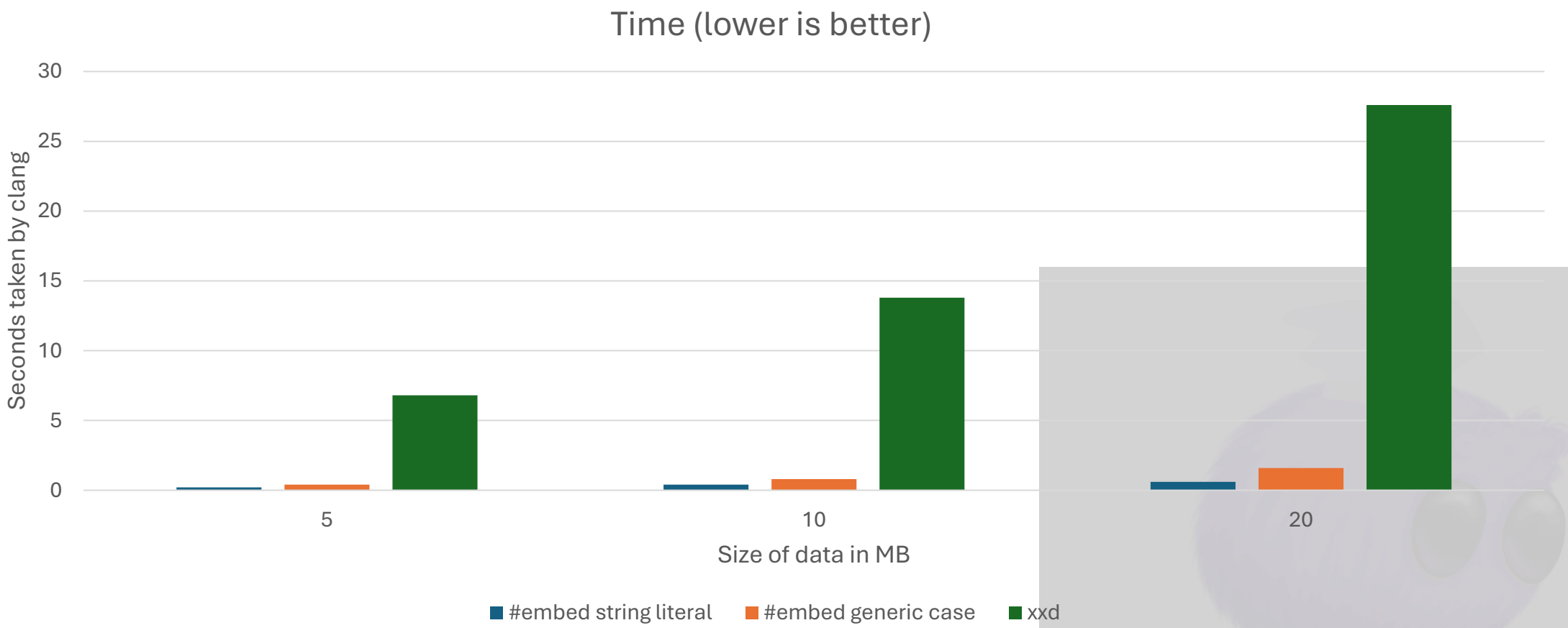
```
// Generic case
int c[] = {1,
#embed "file.bin"
};
```

```
// String literal case
unsigned char b[] = {
#embed "file.bin"
};
```

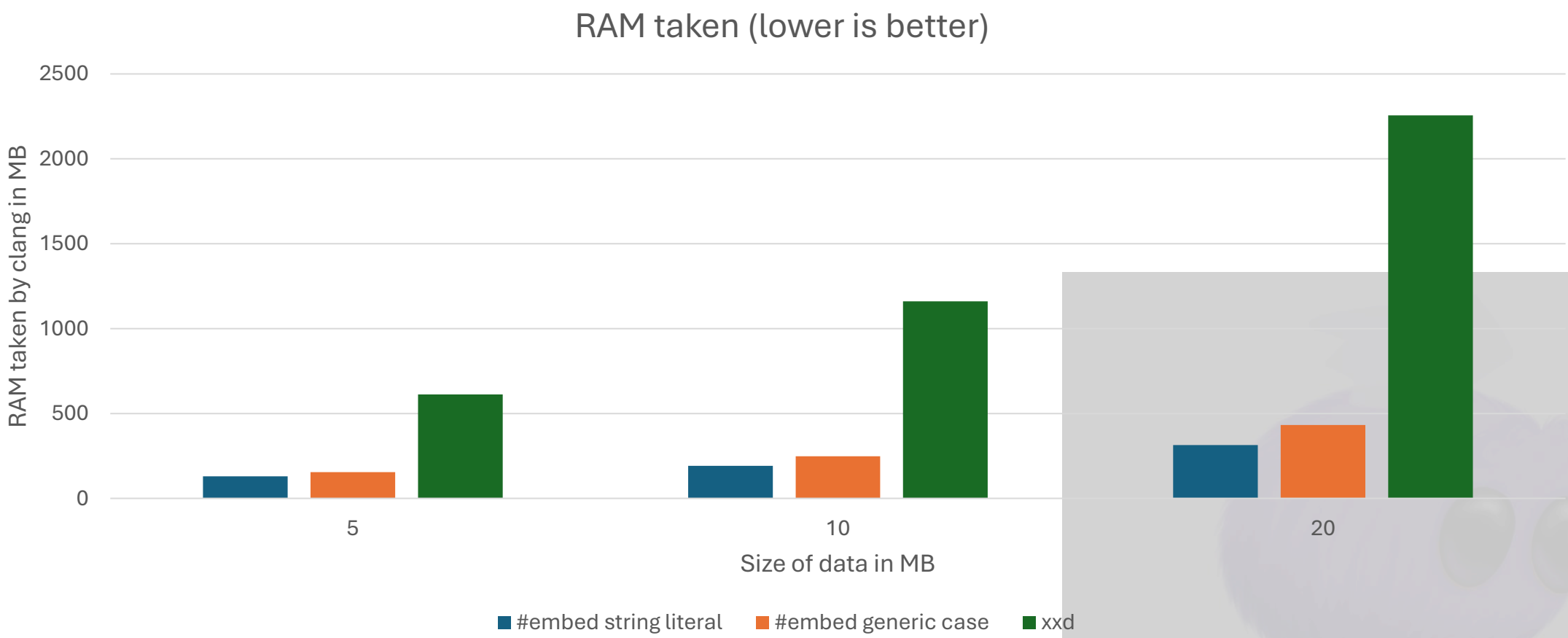
Time difference



Time difference (with xxd)



RAM consumption difference (with xxd)



What is EmbedExpr?

- A reference to embedded data.
- Knows where to take the data and how many of it.
- Represents multiple tokens of data with a single expression.
- One InitListExpr may have several EmbedExpr referencing the same array of data but different parts of this array.
- Created only inside of InitListExpr.
- Handled by AST consumers similarly to array filler.

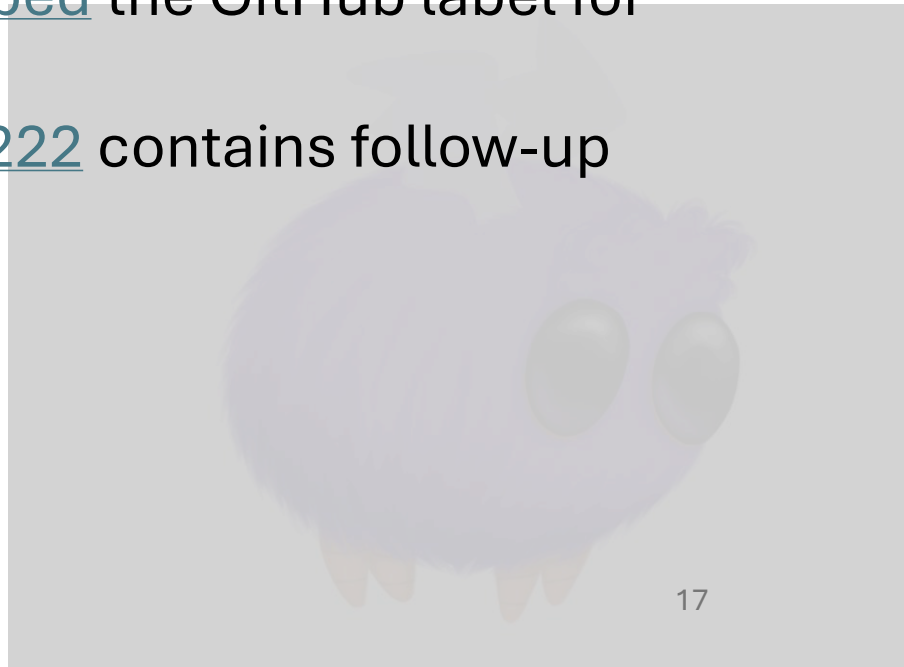
#embed in the wild

```
// 47 is ‘/’
int b = (
#embed __FILE__ limit(2)
);
```

```
`-VarDecl <line:6:1, line:8:1> line:6:5 b 'int'
cinit
  `-ParenExpr <col:9, line:8:1> 'int'
    `-BinaryOperator <line:7:1> 'int' ','
      |-IntegerLiteral <col:1> 'int' 47
      `-IntegerLiteral <col:1> 'int' 47
```

Status in clang

- Available since clang 19.
- Supported in C23, in older C modes and in C++ supported as clang extension.
- Has bugs (known and coming).
  - https://github.com/llvm/llvm-project/labels/embed the GitHub label for #embed-specific bugs.
  - https://github.com/llvm/llvm-project/issues/95222 contains follow-up work to be done/discussed.



Backup

Machine specs

Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz

Ubuntu 24.04

400 GB RAM

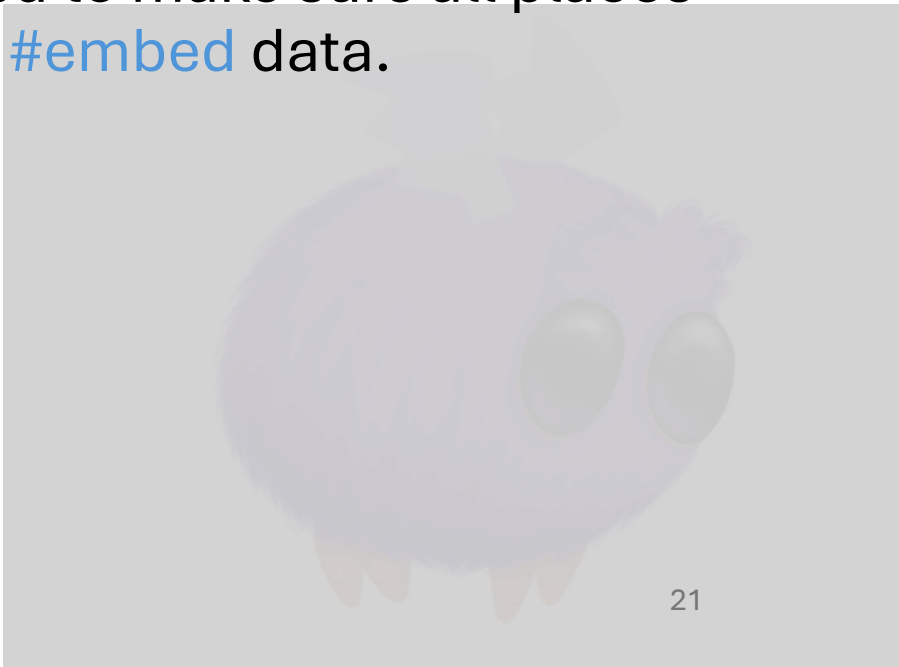
#embed annotation token

```
const int self[] = {
  #embed __FILE__ prefix(1,)
};
```

```
int 'int'       [LeadingSpace]       Loc=<<source>:1:7>
identifier 'self'      [LeadingSpace]
        Loc=<<source>:1:11>
l_square '['           Loc=<<source>:1:15>
r_square ']'           Loc=<<source>:1:16>
equal '='       [LeadingSpace]       Loc=<<source>:1:18>
l_brace '{'     [LeadingSpace]       Loc=<<source>:1:20>
numeric_constant '1'           Loc=<<source>:2:26>
comma ','              Loc=<<source>:2:27>
annot_embed            Loc=<<source>:2:3>
r_brace '}'            Loc=<<source>:3:1>
semi ';'               Loc=<<source>:3:2>
```

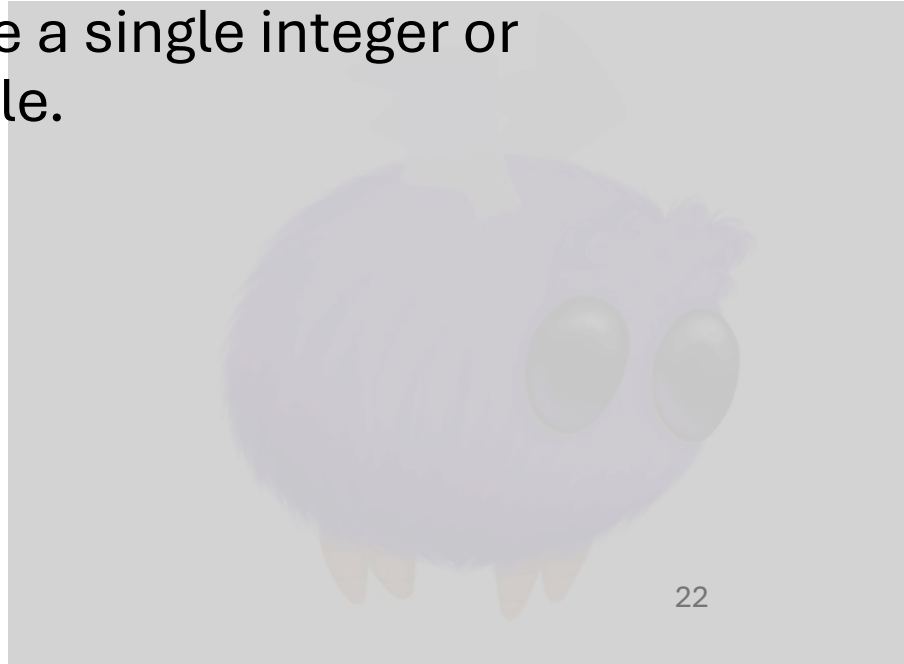
Implementation challenges

- Performance.
  - #embed is easy to implement so it conforms to the standard, yet it is hard to make it effective.
- Corner cases of it being a preprocessor directive.
  - Can output multiple tokens per byte of data. Need to make sure all places where comma-separated list can appear handle #embed data.
- Preprocessed output.
  - -E output can get huge because of #embed.
  - Security concerns.



Why #embed?

- Gets binary content easily into applications.
- Platform independent, portable.
- Allows to include data as a constant expression.
- File search mechanism works like well-known #include directive.
- An #embed directive can be used in any place where a single integer or comma-separated list of integer literals is acceptable.
- Part of C23 standard, accepted in C++26.



Developers' Meeting

BERLIN 2025

AMD

LLVM_ENABLE_RUNTIMES=flang-rt

EuroLLVM 2025

Michael Kruse

Advanced Micro Devices GmbH

15th April, 2025

Outline

Outline

LLVM_ENABLE_RUNTIMES?

Bootstrapping-Runtimes build

```
cmake ../llvm -GNinja                                          \
  "-DLLVM_ENABLE_PROJECTS=clang;lld;polly"                     \
  "-DLLVM_ENABLE_RUNTIMES=compiler-rt;libc;libcxx;openmp"
```

LLVM_ENABLE_PROJECTS

- Compiled using host compiler (e.g. GCC)
- Same CMake build-dir as LLVM itself
- Intended to run on the host architecture

LLVM_ENABLE_RUNTIMES

- Compiled using just-built Clang
- Use separate CMake build-dir nested inside LLVM build-dir
- Intended to be used by binaries compiled by Clang
  - Can be a different architecture (cross-compilation)

How does it work?

CMake step

LLVM_ENABLE_PROJECTS

1. For each enabled project,
   `add_subdirectory(llvm-sourcedir/<project>)`

LLVM_ENABLE_RUNTIMES

1. Add build targets:
   `runtimes`, `install-runtimes`, `<runtime>`, `check-<runtime>`, `install-<runtime>`, …
2. For each target architecture,

`llvm_ExternalProject_Add(runtimes ...)`

which executes

```
cmake -GNinja                                          \
  -S llvm-sourcedir/runtimes                           \
  -B llvm-builddir/runtimes/runtimes-bins              \
  -DLLVM_BINARY_DIR=llvm-builddir                      \
  -DCMAKE_{C,CXX}_COMPILER=llvm-builddir/bin/clang{++} \
  -DCMAKE_{C,CXX}_COMPILER_WORKS=YES                   \
  "-DLLVM_ENABLE_RUNTIMES=<runtimes>"
```

1. `find_package(LLVM)`, `find_package(Clang)`
2. Find tools such as `llvm-lit`, `FileCheck` in `llvm-builddir`
3. For each enabled runtime,
   `add_subdirectory(llvm-sourcedir/<runtime>)`

How does it work?

Ninja step

LLVM_ENABLE_PROJECTS: `ninja <project>`

1. CMake ensured all necessary dependencies

LLVM_ENABLE_RUNTIMES: `ninja <runtime>`

1. Build dependencies such as `clang`, `FileCheck`, etc
2. Run configure step for runtimes
3. Build dependencies for runtimes
4. Execute

`ninja -C llvm-builddir/runtimes/runtime-bins <runtime>`

    1. Build selected runtime

Runtime Options

Pass Options to Runtimes Build

```
cmake ...                                                    \
  -DCMAKE_CXX_FLAGS=-fmax-errors=1                           \
  "-DRUNTIMES_CMAKE_ARGS=-DCMAKE_CXX_FLAGS=-ferror-limit=1"
```

- `-fmax-errors=1` is for gcc
- `-ferror-limit=1` is for clang

Multiarch & Pass Arch-specific Options

```
cmake ...                                                      \
  "-DLLVM_RUNTIME_TARGETS=default;aarch64-linux-gnu"            \
  "-DRUNTIMES_aarch64-linux-gnu_CMAKE_CXX_FLAGS=-march=cortex-a57"
```

- Will create one builddir per target

Standalone-Runtimes build

CMake step

```
cmake -GNinja llvm-srcdir/runtimes      \
  -DLLVM_BINARY_DIR=llvm-builddir       \
  "-DLLVM_ENABLE_RUNTIMES=<runtimes>"
```

- Projects (LLVM, Clang, ...) compiled separately
- Uses default C/C++ compiler (e.g. gcc)

Ninja step

```
ninja <runtime>
ninja check-<runtime>
```

Outline

Legacy Flang Runtime

flang/runtime/CMakeLists.txt

In-Tree build

- Like a LLVM_ENABLE_RUNTIMES build:

```
add_subdirectory(runtime)
```

Standalone build

```
cmake llvm-srcdir/flang/runtime \
  -DLLVM_DIR=...                \
  -DCLANG_DIR=...               \
  -DMLIR_DIR=...

cmake llvm-srcdir/flang/lib/Decimal \
  -DLLVM_DIR=...                    \
  -DCLANG_DIR=...                   \
  -DMLIR_DIR=...
```


What is the Problem?

- Inconsistent with other LLVM runtimes
- For cross-compilation targets, must compile each target in standalone build
- GPU offloading: Build auxiliary target runtime separately
  - As done for openmp-offload, libc, compiler-rt, libcxx, ...
- Standalone build does not include `iso_fortran_env_impl.f90`
- Source code shared with Flang and Runtime
  - ABI assumed to be the same
  - Compile code and runtime code have different requirements
    E.g. runtime code must not link to C++ standard library
  - No clear separation which file belongs where
- Compiled binary shared with Flang and Runtime
  - Runtime built with different flags
    E.g. `-fno-lto`

Outline

Building Flang-RT

Bootstrapping-Runtimes build

```
cmake  -GNinja ../llvm                        \
  "-DLLVM_ENABLE_PROJECTS=clang;mlir;flang" \
  "-DLLVM_ENABLE_RUNTIMES=flang-rt"
```

Standalone-Runtimes build

```
cmake -GNinja llvm-srcdir/runtimes                  \
  "-DLLVM_ENABLE_RUNTIMES=flang-rt"                 \
  -DLLVM_BINARY_DIR=llvm-builddir                   \
  -DCMAKE_Fortran_COMPILER=llvm-builddir/bin/flang \
  -DCMAKE_Fortran_COMPILER_WORKS=YES
```

- Flang must built from the same git SHA1
  No ABI contract
- `CMAKE_Fortran_COMPILER_WORKS` because flang before the runtime is available cannot produce executables

Things that Must Continue Working

- Shared library
- Quad-precision `math.h` support
  - gcc `libquadmath`
  - Native `sizeof(long double) == 16` with `libm`
  - f128 suffix functions (like `sinf128`) in `libm`
- Conditional `REAL(16)` support in Flang
- Unittests
  - GTest and "non-gtest" testing framework
- Windows static `.lib`
  - LLVM emits libgcc-ABI function calls, requires `clang_rt.builtins.lib` at link-time
  - msvc ships `clang_rt.builtins-x86_64.a`, but not used by the driver (anymore)
- Experimental OpenMP-offload build
- Experimental CUDA build
  - With `clang -x cuda` and `nvcc`

Library Names

Old Library Names

- `libFortranRuntime{.a,.so}`
- `libFortranDecimal{.a,.so}`
- `libFortranFloat128Math.a`
- `libCufRuntime_cuda_${version}{.a,.so}`

New Library Names

- `libflang_rt.runtime{.a,.so}`
- `libflang_rt.quadmath.a`
- `libflang_rt.cuda_${version}{.a,.so}`

- Same scheme as Compiler-RT: `libclang_rt.<component>{.a,.so}`
- `libFortranDecimal` integrated into `libflang_rt.runtime`
  - Decided by RFC
  - `libFortranCommon` also used by Flang
  - Made `flang` depend on `libcudart.so`

The Big Move

Principles

- Split some headers into a compiler- and a runtime part
- Definitions to `flang-rt/lib/$component/*.cpp`
- Non-private headers to `flang-rt/include/flang-rt/$component/*.h`
- Files used by both, Flang (the compiler) and Flang-RT (the runtime), remain in `flang/`
- Move "Common" files only used by Flang (the compiler) to `Support`
  - Remaining shared components: `FortranDecimal`, `FortranCommon` (header-only), `FortranRuntime` (header-only), `FortranTesting`

Old	New
`flang/runtime/*.h`	`flang-rt/include/runtime/*.h`
`flang/runtime/*.cxx`	`flang-rt/lib/runtime/*.cpp`
`flang/runtime/Float128Math/*`	`flang-rt/lib/quadmath/*`
`flang/runtime/CUDA/*`	`flang-rt/lib/cuda/*`
`flang/include/flang/Runtime/*.h`	`flang/include/flang/Runtime/*.h`
`flang/include/flang/Common/*.h`[1]	`flang/include/flang/Support/*.h`
`flang/unittests/Evaluate/{fp-}testing.h`	`flang/include/flang/Testing/*.h`
`flang/lib/Common/*.cpp`	`flang/lib/Support/*.cpp`
`flang/unittests/Evaluate/{fp-}testing.cpp`	`flang/lib/Testing/*.cpp`
`flang/test/**/*`[2]	`flang-rt/test/**/*.cpp`

LLVM_ENABLE_PER_TARGET_RUNTIME_DIR

Library Location

- Old Flang Runtime:
  `${CMAKE_INSTALL_PREFIX}/lib/libflang_rt.runtime.a`
  - Clash in multiarch/cross-compile scenarios
- `LLVM_ENABLE_PER_TARGET_RUNTIME_DIR=OFF`:
  `${CMAKE_INSTALL_PREFIX}/lib/clang/$version/lib/$os/libclang_rt.buildins-$arch.a`
  - Windows, Apple, AIX
- `LLVM_ENABLE_PER_TARGET_RUNTIME_DIR=ON`:
  `${CMAKE_INSTALL_PREFIX}/lib/clang/$version/lib/$triple/libclang_rt.builtins.a`
  - Became default for Linux in Clang 19 (Now also BSD, OS390)
  - Assumptions leaking into `LLVM_ENABLE_PER_TARGET_RUNTIME_DIR=OFF` as well 😕

- Only the last supported for flang-rt
  - `${CMAKE_INSTALL_PREFIX}/lib/clang/$version/lib/$triple/libflang_rt.<component>{.a,.so}`
  - `LLVM_ENABLE_PER_TARGET_RUNTIME_DIR` ignored

Shared Library

- Old scheme: `BUILD_SHARED_LIBS=ON`
  - Requires a second standalone build
- New scheme: Build static+shared library at the same time using *object libraries*
  - Done by (almost) every other runtime

CMake Options

- `FLANG_RT_ENABLE_STATIC`
  - Default: `ON`
- `FLANG_RT_ENABLE_SHARED`
  - Default: `OFF`

  ld prefers `.so` over `.a`, enabling it would be a breaking change

Experimental GPU Target Support

CUDA

- **clang**: Compile everything with `-x cuda`
- **nvcc**: Treats everything as CUDA source
- Requires libcudac++ (libc++ for CUDA), cannot use `<variant>` or `<optional>`
- Declarations must be annotated with `__host__ __device__`

OpenMP

- Compile everything with `-fopenmp --offload-arch`
- Declarations must be annotated with `#pragma declare target`

- Annotations selected with preprocessor macro `RT_API_ATTRS`
- Results in a *fat library*
  - Host and device code in a single file
  - For AMD/OpenMP we would rather compile them separately
    - Multiarch library with device code looked up when launching kernels

Outline

To Do Items

- Flang is not (yet) a cross-compiler

  Sometimes assumes ABI of host platform, e.g. `sizeof(long)`
- Compile builtin modules in the runtimes build using CMake
  - OpenMP's modules as well
  - Per-target modules
- Flang's Quadmath support must not depend on LLVM build environment
- Multilib support
- Shared library location
- Library versioning


AMD

Developers' Meeting

BERLIN 2025

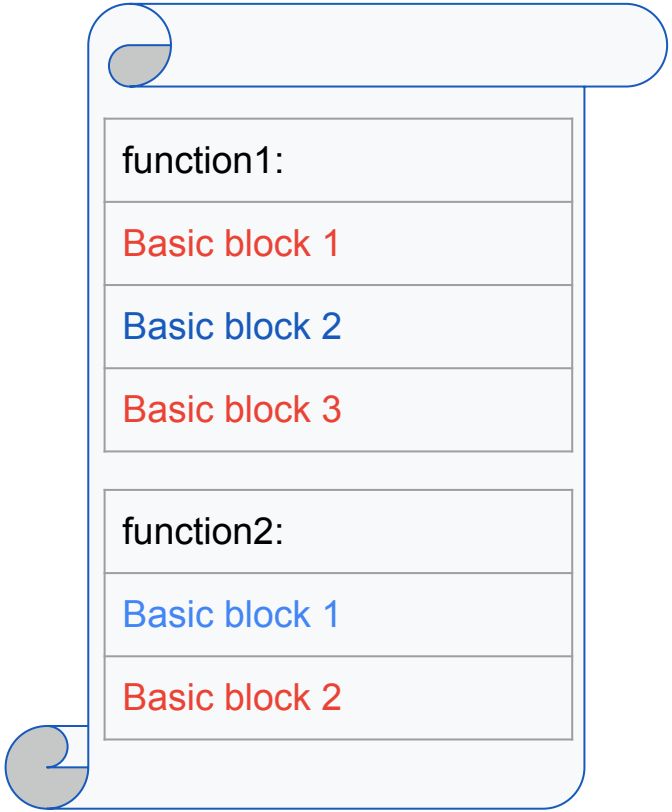
Google

LLDB support for Propeller optimized code

(things programmers believe about functions)

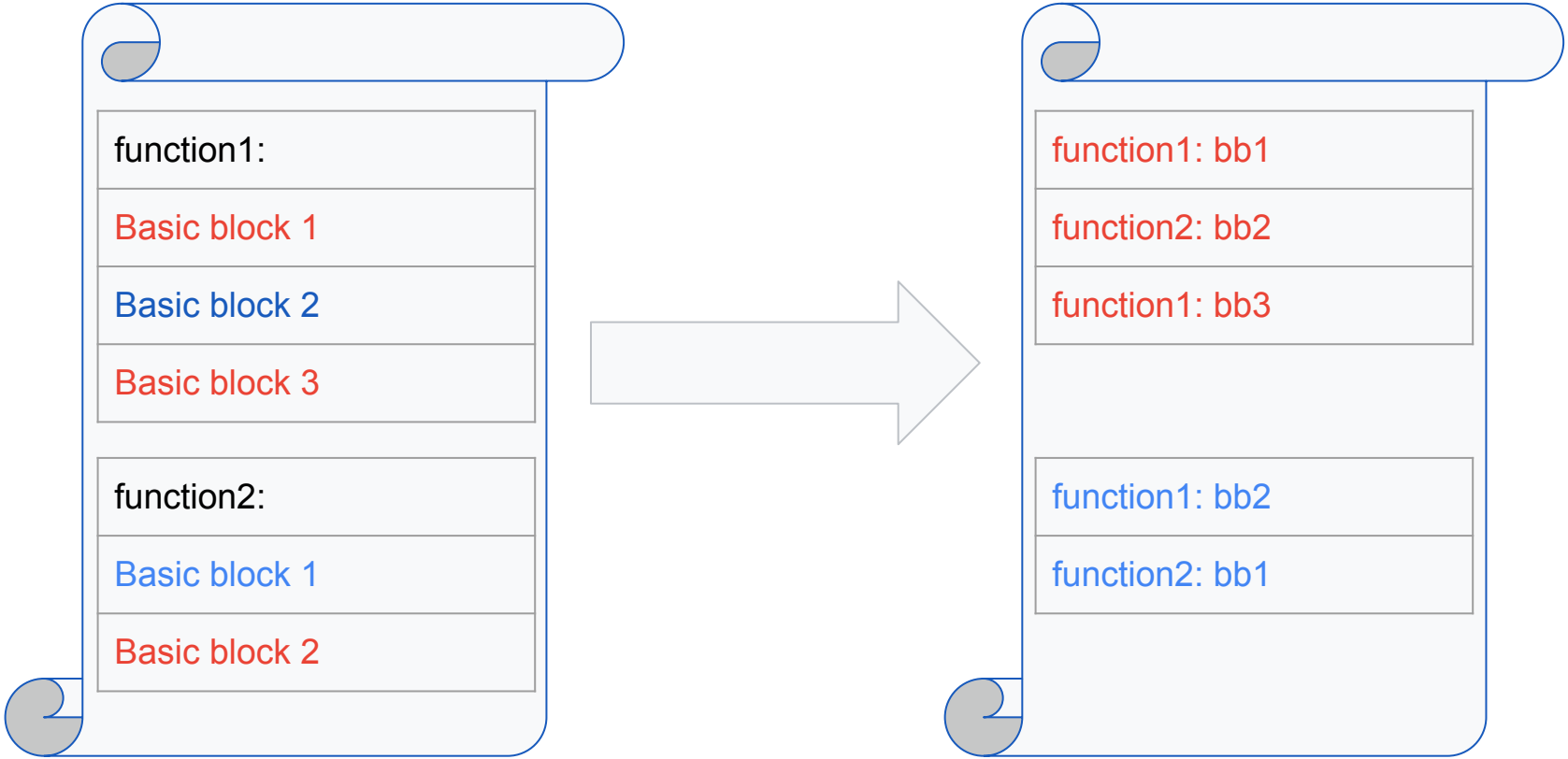
Pavel Labath

Propeller (Bolt, etc.)



Google

Propeller (Bolt, etc.)



Google

Debug info (DWARF)

```
DW_TAG_subprogram
    DW_AT_name    ("function1")
    DW_AT_low_pc  (0x00001000)
    DW_AT_high_pc
(0x00001030)
    ...
DW_TAG_subprogram
    DW_AT_name    ("function2")
    DW_AT_low_pc  (0x00002000)
    DW_AT_high_pc
(0x00002020)
    ...
```

Debug info (DWARF)

```
DW_TAG_subprogram
    DW_AT_name    ("function1")
    DW_AT_low_pc  (0x00001000)
    DW_AT_high_pc
(0x00001030)
    ...
DW_TAG_subprogram
    DW_AT_name    ("function2")
    DW_AT_low_pc  (0x00002000)
    DW_AT_high_pc
(0x00002020)
    ...
```

```
DW_TAG_subprogram
    DW_AT_name    ("function1")
    DW_AT_ranges  (
        [0x00001000, 0x00001010)
        [0x00002000, 0x00002010)
        [0x00001020, 0x00001030))
    ...
DW_TAG_subprogram
    DW_AT_name    ("function2")
    DW_AT_ranges  (
        [0x00002010, 0x00002020)
        [0x00001010, 0x00001020))
    ...
```

Google

The problem

```
class Function {
    const AddressRange &GetAddressRange() { return m_range; }
    ...
};

function1.GetAddressRange() = [0x1000, 0x2010)
function2.GetAddressRange() = [0x1010, 0x2020)
```

The problem

```
class Function {
    const AddressRange &GetAddressRange() { return m_range; }
    ...
};

function1.GetAddressRange() =  [0x1000, 0x2010)
function2.GetAddressRange() =  [0x1010, 0x2020)

function1.GetAddressRange().GetBaseAddress() =  0x1000
function2.GetAddressRange().GetBaseAddress() =  0x1010
```

The problem

```
class Function {
    const AddressRange &GetAddressRange() { return m_range; }
    ...
};

function1.GetAddressRange() = [0x1000, 0x2010)
function2.GetAddressRange() = [0x1010, 0x2020)

function1.GetAddressRange().GetBaseAddress() = 0x1000
function2.GetAddressRange().GetBaseAddress() = 0x1010

class SymbolContext {
    bool GetAddressRange(uint32_t scope, uint32_t range_idx,
        bool use_inline_block_range, AddressRange &range) const;
};
```

Google

Solution

```
class Function {
    AddressRanges GetAddressRanges() { return m_block.GetRanges(); }
    const Address &GetAddress() const { return m_address; }
};


class SymbolContext {
    bool GetAddressRange(uint32_t scope, uint32_t range_idx,
        bool use_inline_block_range, AddressRange &range) const;
    Address GetFunctionOrSymbolAddress() const;
};
```

Google

Recommendations

- Do not assume that functions are contiguous

Recommendations

- Do not assume that functions are contiguous
- Do not assume that all parts (address ranges) of the function are within a single section
  - ```
    if (addr1.GetSection() == addr2.GetSection())
        return addr1.GetOffset() - addr2.GetOffset();
    ```
  - ```
    if (addr1.GetModule() == addr2.GetModule())
        return addr1.GetFileAddress() - addr2.GetFileAddress();
    ```

Recommendations

- Do not assume that functions are contiguous
- Do not assume that all parts (address ranges) of the function are within a single section
  - ```
    if (addr1.GetSection() == addr2.GetSection())
        return addr1.GetOffset() - addr2.GetOffset();
    ```
  - ```
    if (addr1.GetModule() == addr2.GetModule())
        return addr1.GetFileAddress() - addr2.GetFileAddress();
    ```
- Do not assume that a function is described by a single `eh_frame` record (line table sequence, etc.)

Google

Recommendations

- Do not assume that functions are contiguous
- Do not assume that all parts (address ranges) of the function are within a single section
  - ```
    if (addr1.GetSection() == addr2.GetSection())
        return addr1.GetOffset() - addr2.GetOffset();
    ```
  - ```
    if (addr1.GetModule() == addr2.GetModule())
        return addr1.GetFileAddress() - addr2.GetFileAddress();
    ```
- Do not assume that a function is described by a single `eh_frame` record (line table sequence, etc.)
- Do not assume that function entry point is its lowest address

Recommendations

- Do not assume that functions are contiguous
- Do not assume that all parts (address ranges) of the function are within a single section
  - ```
    if (addr1.GetSection() == addr2.GetSection())
        return addr1.GetOffset() - addr2.GetOffset();
    ```
  - ```
    if (addr1.GetModule() == addr2.GetModule())
        return addr1.GetFileAddress() - addr2.GetFileAddress();
    ```
- Do not assume that a function is described by a single `eh_frame` record (line table sequence, etc.)
- Do not assume that function entry point is its lowest address
  - Corollary: Do not assume that offsets from the entry point are positive

Current state

- Most things “just work”
- Main exception: unwinding

Current state

- Most things “just work”
- Main exception: unwinding
- There are rough edges:

```
(lldb) disassemble --name foo
my_binary`foo:
0xdead00 <-1179648>: cmpl   $0x0, %eax
0xdead03 <-1179645>: jmp    0xdead33        ; <-1179597>
...
```

Thank you

Developers' Meeting

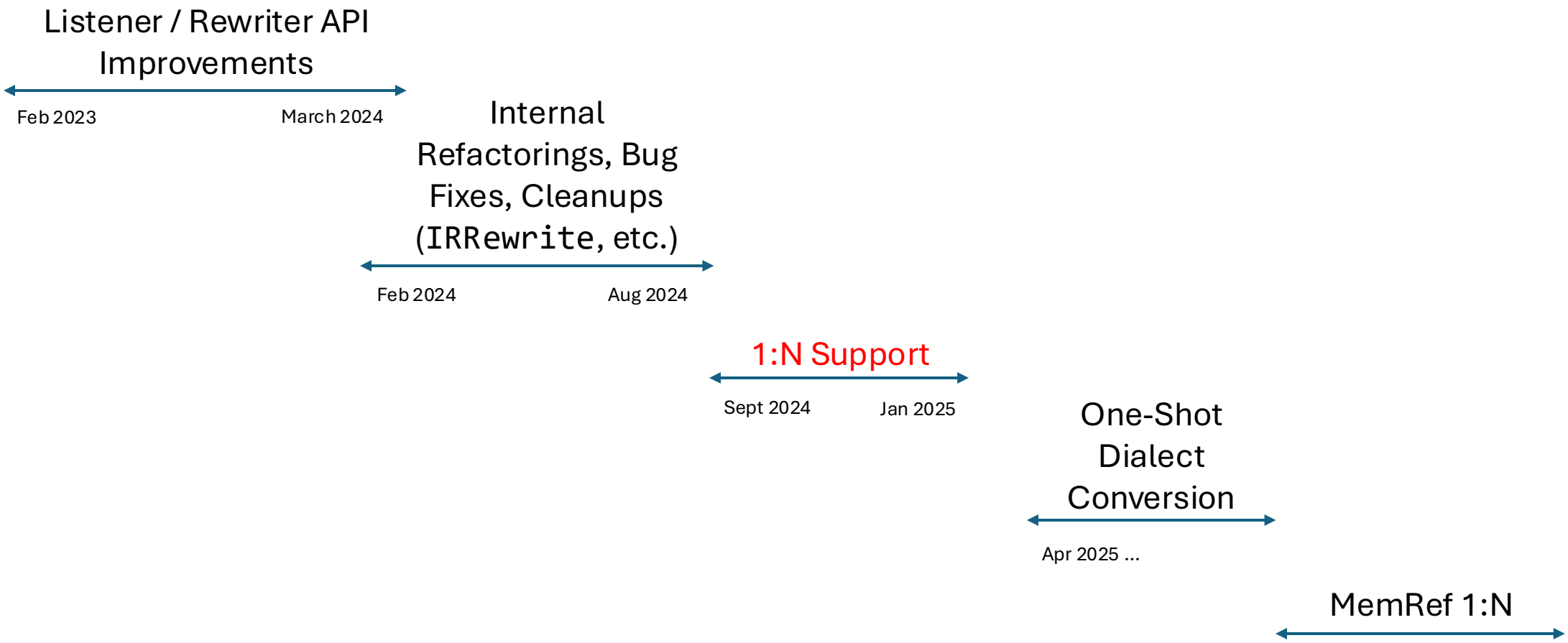
BERLIN 2025

1:N Dialect Conversion

Matthias Springer (NVIDIA)

EuroLLVM 2025 – Quick Talk – April 15, 2025

Timeline


Listener / Rewriter API Improvements
Feb 2023
March 2024
Internal Refactorings, Bug Fixes, Cleanups (IRRewrite, etc.)
Feb 2024
Aug 2024
1:N Support
Sept 2024
Jan 2025
One-Shot Dialect Conversion
Apr 2025 ...
MemRef 1:N

Dialect Conversion

- One of the two main pattern drivers in MLIR, more powerful API than “regular” rewrite API (`PatternRewriter`)

`PatternRewriter`	`ConversionPatternRewriter`
`replaceOp(Operation *, ValueRange);`	`replaceOp(Operation *, ValueRange);`
a combination of: `createBlock`, `inlineBlockBefore`, create unrealized_conversion_cast ops, `replaceAllUsesWith`, `eraseBlock`	`applySignatureConversion(Block *, SignatureConversion &, TypeConverter &);`

Dialect Conversion

- One of the two main pattern drivers in MLIR, more powerful API than “regular” rewrite API (`PatternRewriter`)

`PatternRewriter`	`ConversionPatternRewriter`
`replaceOp(Operation *, ValueRange);`	`replaceOp(Operation *, ValueRange);`
a combination of: `createBlock`, `inlineBlockBefore`, create unrealized_conversion_cast ops, `replaceAllUsesWith`, `eraseBlock`	`applySignatureConversion(Block *, SignatureConversion &, TypeConverter &);`

For each block argument, specifies the new type in the converted block.

Dialect Conversion

- One of the two main pattern drivers in MLIR, more powerful API than “regular” rewrite API (`PatternRewriter`)
- What’s new: 1:N op replacements, 1:N conversion patterns

PatternRewriter	ConversionPatternRewriter
`replaceOp(Operation *, ValueRange);`	`replaceOp(Operation *, ValueRange);` `replaceOpWithMultiple(Operation *, ArrayRef<ValueRange>);`
a combination of: `createBlock`, `inlineBlockBefore`, create unrealized_conversion_cast ops, `replaceAllUsesWith`, `eraseBlock`	`applySignatureConversion(Block *, SignatureConversion &, TypeConverter &);`

For each block argument, specifies the new types in the converted block.

Outline of This Work

- **Cleanup** of duplicate + non-composable frameworks:
  - Incomplete implementation: 1:N used to be partially supported in the dialect conversion framework (only signature conversions).
  - There is a separate 1:N dialect conversion framework that supports 1:N op replacements, but it's not really a dialect conversion.
- Why is this important?
  - 1:N conversions appear in **load-bearing + performance critical** lowering passes of real-world compilers.
    - MLIR: MemRef → LLVM Lowering
    - MLIR: Sparse Tensor → Sparse Tensor Descriptor (codegen path)
    - Triton: Tile → SIMT
    - Multiple NVIDIA-internal projects: cuTile, Cutlass, …
  - These passes must resort to packing/unpacking to work around dialect conversion limitations. That's **inefficient** (increases compilation time) and makes code/IR **complex**.

Example: MemRef → LLVM

Example: 1:1 MemRef → LLVM Lowering

```
memref<?x?xf32, strided<[?, ?], offset: ?>>
```

1 SSA value → 1 SSA value

```
!llvm.struct<(!llvm.ptr, !llvm.ptr,       // ptr
              i64,                        // offset
              !llvm.struct<(i64, i64)>,   // sizes
              !llvm.struct<(i64, i64)>)   // strides
```

Example: 1:N MemRef → LLVM Lowering

```
memref<?x?xf32, strided<[?, ?], offset: ?>>
```

1 SSA value → 7 SSA values

```
!llvm.ptr, !llvm.ptr,                    // ptr
i64,                                     // offset
i64, i64                                 // sizes
i64, i64                                 // strides
```

Example: 1:1 MemRef → LLVM Lowering

```
memref<*xf32>
```

1 SSA value → 1 SSA value

```
!llvm.struct<(!llvm.ptr,    // ptr to descriptor
              i64)>         // rank
```

Example: 1:N MemRef → LLVM Lowering

```
memref<*xf32>
```

1 SSA value → 2 SSA values

```
llvm.ptr,                    // ptr to descriptor
i64                          // rank
```

Test Case

```
// RUN: mlir-opt %s -expand-strided-metadata -convert-to-llvm

func.func @test_case(%m: memref<5xf32>, %offset: index) -> f32 {
    %c1 = arith.constant 1 : index
    %0 = memref.subview %m[%offset][2][1] : memref<5xf32> to memref<2xf32, strided<[1], offset: ?>>
    %1 = memref.load %0[%c1] : memref<2xf32, strided<[1], offset: ?>>
    return %1 : f32
}
```

Test Case

```
// RUN: mlir-opt %s -convert-to-llvm

func.func @test_case(%m: memref<5xf32>, %offset: index) -> f32 {
  %c1 = arith.constant 1 : index
  %base_buffer, %offset, %sizes, %strides = memref.extract_strided_metadata %m
    : memref<5xf32> -> memref<f32>, index, index, index
  %reinterpret_cast = memref.reinterpret_cast %base_buffer to offset: [%offset], sizes: [2], strides: [1]
    : memref<f32> to memref<2xf32, strided<[1], offset: ?>>
  %0 = memref.load %reinterpret_cast[%c1] : memref<2xf32, strided<[1], offset: ?>>
  return %0 : f32
}
```

34 operations

Test Case: 1:1 Lowering (current lowering)

```
llvm.func @test_case(%arg0: !llvm.ptr, %arg1: !llvm.ptr,
                     %arg2: i64, %arg3: i64, %arg4: i64,
                     %arg5: i64) -> f32 {
  %0 = llvm.mlir.poison
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %1 = llvm.insertvalue %arg0, %0[0]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %2 = llvm.insertvalue %arg1, %1[1]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %3 = llvm.insertvalue %arg2, %2[2]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %4 = llvm.insertvalue %arg3, %3[3, 0]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %5 = llvm.insertvalue %arg4, %4[4, 0]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %6 = llvm.mlir.constant(1 : index) : i64
  %7 = llvm.extractvalue %5[0]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %8 = llvm.extractvalue %5[1]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %9 = llvm.mlir.poison : !llvm.struct<(ptr, ptr, i64)>
  %10 = llvm.insertvalue %7, %9[0] : !llvm.struct<(ptr, ptr, i64)>
  %11 = llvm.insertvalue %8, %10[1] : !llvm.struct<(ptr, ptr, i64)>
  %12 = llvm.mlir.constant(0 : index) : i64
  %13 = llvm.insertvalue %12, %11[2] : !llvm.struct<(ptr, ptr, i64)>
  %14 = llvm.extractvalue %5[2]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %15 = llvm.extractvalue %5[3, 0]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %16 = llvm.extractvalue %5[4, 0]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %17 = llvm.mlir.poison
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %18 = llvm.extractvalue %13[0] : !llvm.struct<(ptr, ptr, i64)>
  %19 = llvm.extractvalue %13[1] : !llvm.struct<(ptr, ptr, i64)>
  %20 = llvm.insertvalue %18, %17[0]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %21 = llvm.insertvalue %19, %20[1]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %22 = llvm.insertvalue %arg5, %21[2]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %23 = llvm.mlir.constant(2 : index) : i64
  %24 = llvm.insertvalue %23, %22[3, 0]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %25 = llvm.mlir.constant(1 : index) : i64
  %26 = llvm.insertvalue %25, %24[4, 0]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %27 = llvm.extractvalue %26[1]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %28 = llvm.extractvalue %26[2]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %29 = llvm.getelementptr %27[%28] : (!llvm.ptr, i64) -> !llvm.ptr, f32
  %30 = llvm.getelementptr %29[%6] : (!llvm.ptr, i64) -> !llvm.ptr, f32
  %31 = llvm.load %30 : !llvm.ptr -> f32
  llvm.return %31 : f32
}
```

Test Case: 1:1 Lowering (current lowering)

```
llvm.func @test_case(%arg0: !llvm.ptr, %arg1: !llvm.ptr,
                     %arg2: i64, %arg3: i64, %arg4: i64,
                     %arg5: i64) -> f32 {
  %0 = llvm.mlir.poison
    : !llvm.struct<(ptr, ptr, i64
  %1 = llvm.insertvalue %arg0, %0[0]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %2 = llvm.insertvalue %arg1, %1[1]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %3 = llvm.insertvalue %arg2, %2[2]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %4 = llvm.insertvalue %arg3, %3[3, 0]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %5 = llvm.insertvalue %arg4, %4[4, 0]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %6 = llvm.mlir.constant(1 : index) : i64
  %7 = llvm
    : !llvm
  %8 = llvm.extractvalue %5[1]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %9 = llvm.m
  %10 = llvm.
  %11 = llvm.insertvalue %8, %10[1] : !llvm.struct<(ptr, ptr, i64)>
  %12 = llvm.mlir.constant(0 : index) : i64
  %13 = llvm.insertvalue %12, %11[2] : !llvm.struct<(ptr, ptr, i64)>
  %14 = llv
    : !llvm
  %15 = llvm.extractvalue %5[3, 0]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %16 = llvm.extractvalue %5[4, 0]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %17 = llv
    : !llvm
  %18 = llvm.extractvalue %13[0] : !llvm.struct<(ptr, ptr, i64)>
  %19 = llvm.extractvalue %13[1] : !llvm.struct<(ptr, ptr, i64)>
  %20 = llvm.
    : !llvm.s
  %21 = llvm.insertvalue %19, %20[1]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %22 = llvm.insertvalue %arg5, %21[2]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %23 = llvm.mlir.constant(2 : index) : i64
  %24 = llvm.insertvalue %23, %22[3, 0]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %25 = llvm.mlir.constant(1 : index) : i64
  %26 = llvm.insertvalue %25, %24[4, 0]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %27 = llvm.extractvalue %26[1]
    : !llvm.struct<(ptr, ptr, i64,
  %28 = llvm.extractvalue %26[2]
    : !llvm.struct<(ptr, ptr, i64, array<1 x i64>, array<1 x i64>)>
  %29 = llvm.getelementptr %27[%28] : (!llvm.ptr, i64) -> !llvm.ptr, f32
  %30 = llvm.getelementptr %29[%6] : (!llvm.ptr, i64) -> !llvm.ptr, f32
  %31 = llvm.load %30 : !llvm.ptr -> f32
  llvm.return %31 : f32
}
```

target materialization: pack

ExtractStridedMetadataOpLowering: unpack

ExtractStridedMetadataOpLowering: pack

ExtractStridedMetadataOpLowering: unpack

MemRefReinterpretCastOpLowering: unpack

MemRefReinterpretCastOpLowering : pack

LoadOpLowering :unpack

13 operations: 61% reduction in number of operations

Test Case: 1:N Lowering (better lowering)

```
llvm.func @bar(%arg0: !llvm.ptr, %arg1: !llvm.ptr, %arg2: i64, %arg3: i64, %arg4: i64, %arg5: i64) -> f32 {
  %0 = llvm.mlir.constant(1 : index) : i64
  %1 = llvm.mlir.poison : !llvm.ptr
  %2 = llvm.mlir.poison : i64
  %3 = llvm.mlir.constant(0 : index) : i64
  %4 = llvm.mlir.poison : !llvm.ptr
  %5 = llvm.mlir.poison : i64
  %6 = llvm.mlir.constant(2 : index) : i64
  %7 = llvm.mlir.constant(1 : index) : i64
  %8 = llvm.getelementptr %arg1[%arg5] : (!llvm.ptr, i64) -> !llvm.ptr, f32
  %9 = llvm.getelementptr %8[%0] : (!llvm.ptr, i64) -> !llvm.ptr, f32
  %10 = llvm.load %9 : !llvm.ptr -> f32
  llvm.return %10 : f32
}
```

Prototype: llvm/llvm-project:users/matthias-springer/memref_1_to_n

1:N Conversion API

Type Converter

These are the types of the `ValueRange` that the adaptor returns.

```
converter.addConversion([&](MemRefType type,
                            SmallVectorImpl<Type> &result) -> std::optional<LogicalResult> {
  result.push_back(llvmPtrTy);              // allocated ptr
  result.push_back(llvmPtrTy);              // aligned ptr
  result.push_back(i64Ty);                  // offset
  for (int64_t i = 0; i < rank; ++i)
    result.push_back(i64Ty);                // sizes
  for (int64_t i = 0; i < rank; ++i)
    result.push_back(i64Ty);                // strides
  return success();
});

// previously: returned !llvm.struct<(…)>
```

Conversion Pattern

```
// Simplified: Assume that source is a ranked memref and index is static.
class DimOpLowering : public OpConversionPattern<memref::DimOp> {
  LogicalResult matchAndRewrite(memref::DimOp op, OneToNOpAdaptor adaptor,
                                ConversionPatternRewriter &rewriter) const override {
    int64_d dim = op.getIndex();
    int64_t descriptorPos = 2 + 1 + dim;
    ValueRange descriptor = adaptor.getSource();
    rewriter.replaceOp(op, descriptor[descriptorPos]);
    return success();
  }
};

// previously: OpAdaptor
```

Conversion Pattern (incorrect example)

```
// Simplified: Assume that source is a ranked memref and index is static.
class DimOpLowering : public OpConversionPattern<memref::DimOp> {
  LogicalResult matchAndRewrite(memref::DimOp op, OpAdaptor adaptor,
                                ConversionPatternRewrit                iter) const override {
    int64_d dim = op.getIndex();
    int64_t descriptorPos = 2 + 1 + dim;
    Value descriptor = adaptor.getSource();
    rewriter.replaceOp(op, ???);
    return success();
  }
};

// previously: OpAdaptor
```

What if I use a regular adaptor in a 1:N conversion?

LLVM fatal error: 'DimOpLowering' does not support 1:N conversion

Note: You can use a regular adaptor if all converted values are guaranteed to be single SSA values.

Migration Path from Deprecated 1:N Dialect Conversion

`OneToNTypeConversion.h`

Migration Guide

- Will be deleted from upstream MLIR after this conference!
- Migrate patterns, driver invocation and tests
  - `OneToNConversionPattern` → `ConversionPattern`
  - `applyPartialOneToNConversion` → `applyPartialConversion`. You're going to have to define a `ConversionTarget`.
  - `replaceOp(Operation *, ValueRange, OneToNTypeMapping)` → `replaceOpWithMultiple(Operation*, ArrayRef<ValueRange>)`
  - `OneToNTypeMapping` → `SignatureConversion`
  - Update test cases: Old framework was actually a greedy pattern rewrite that performs additional CSE'ing, folding, region simplification.

Questions?

`applyPartialConversion`

`applyPartialOneToNConversion`

`applySignatureConversion`

`ConversionPattern`

`ConversionPatternRewriter`

`ConversionTarget`

`LLVMStructType`

`MemRefType`

`OpAdaptor`

`OneToNOpAdaptor`

`OneToNTypeMapping`

`replaceOp`

`replaceOpWithMultiple`

`SignatureConversion`

source materialization

`TypeConverter`

target materialization

`UnrankedMemRefType`

`unrealized_conversion_cast`

Developers' Meeting

BERLIN 2025