#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
```

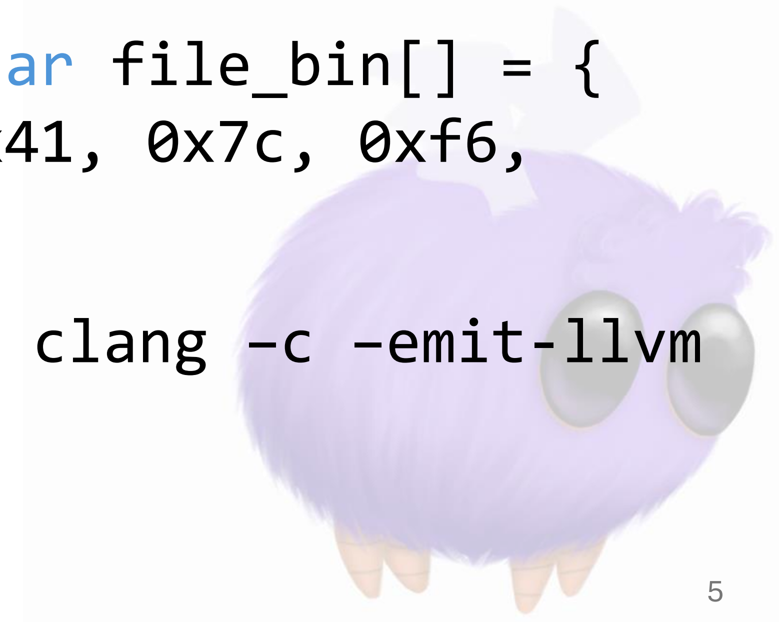
embed.c

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

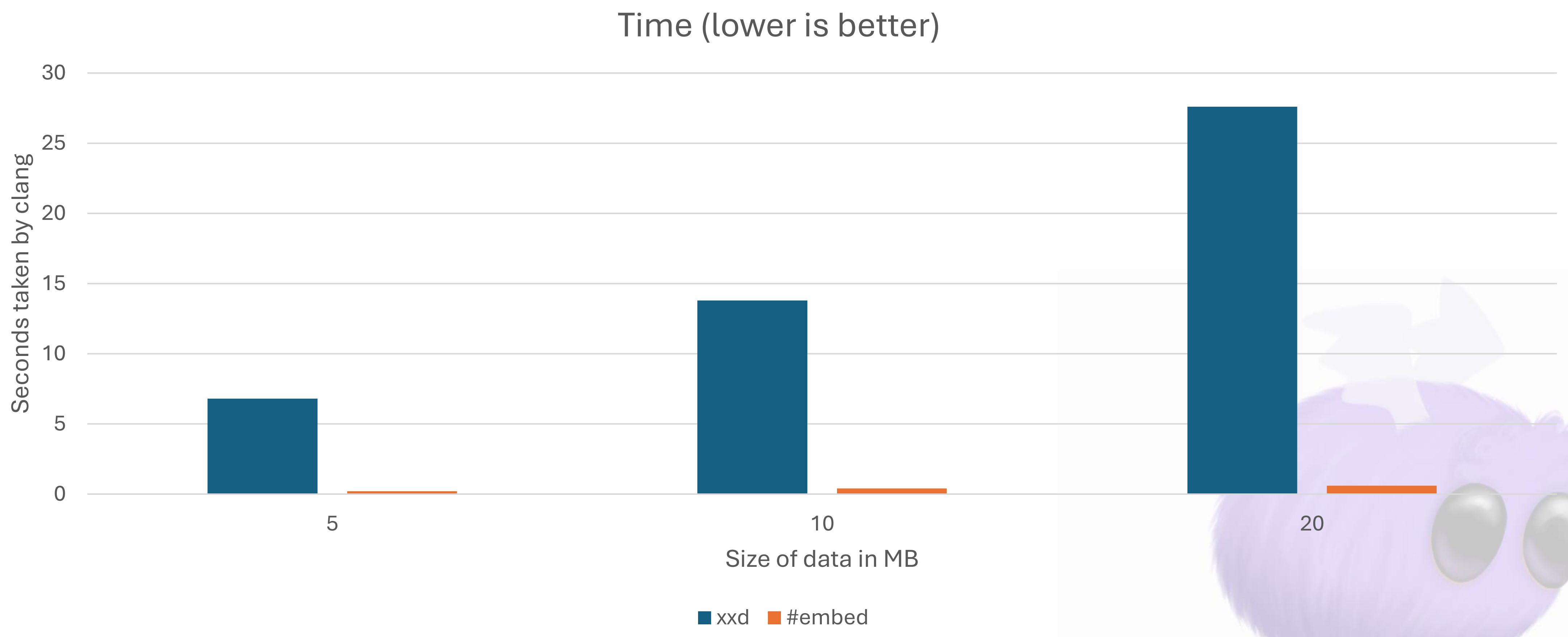
filexxd.c

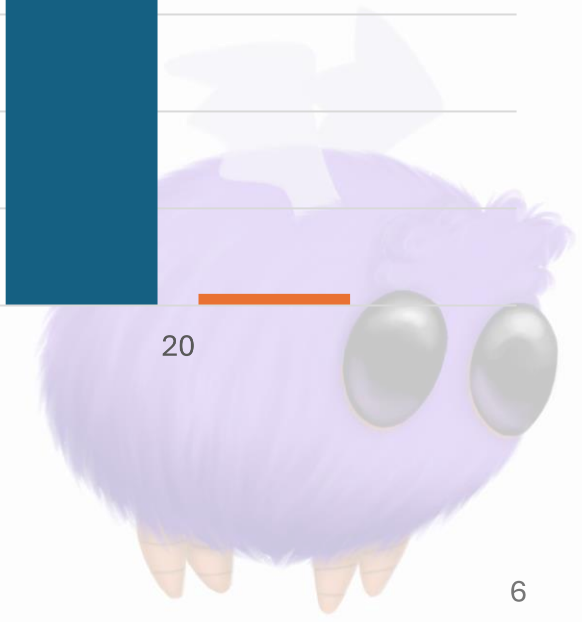
```
unsigned char file_bin[] = {
   0x82, 0x41, 0x7c, 0xf6,
0x7c,…
```

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

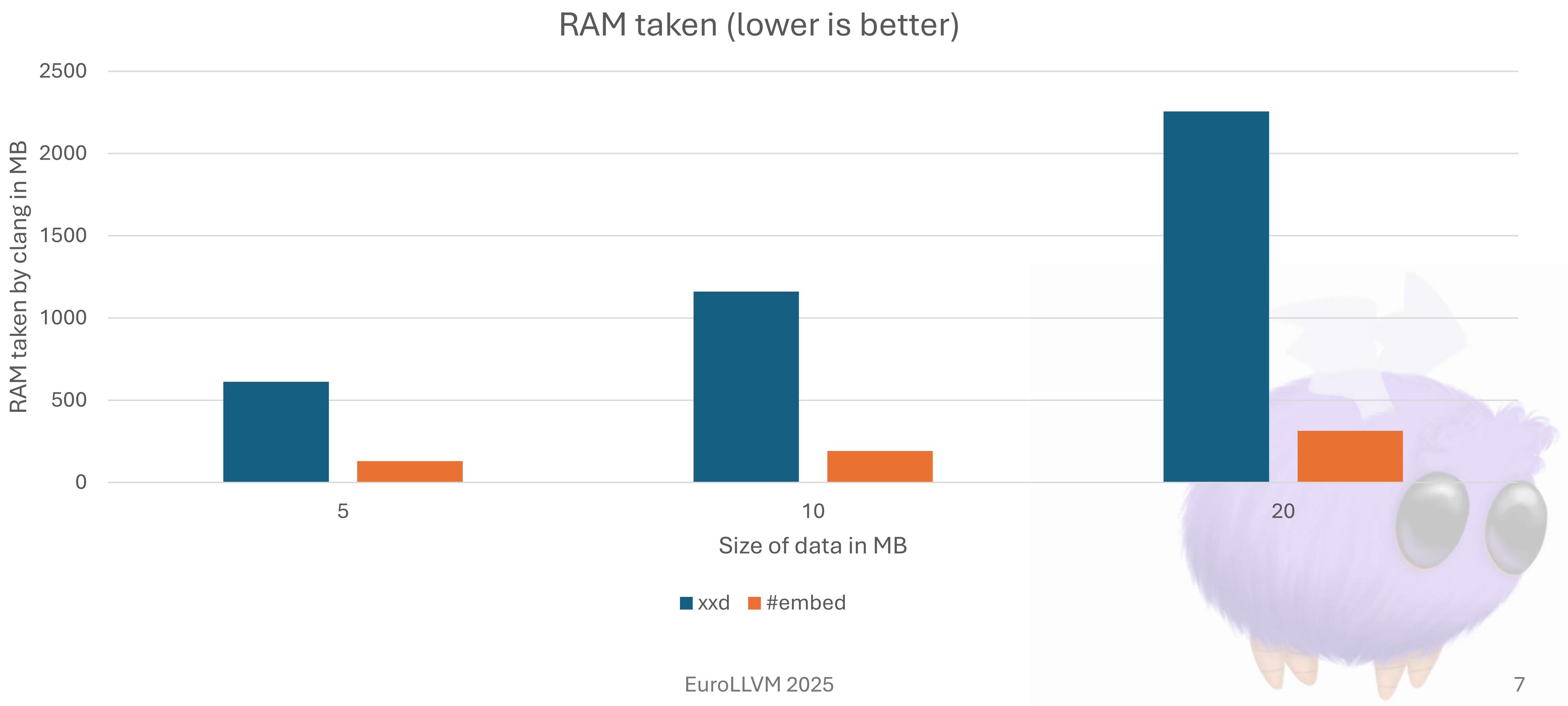


Time difference





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.

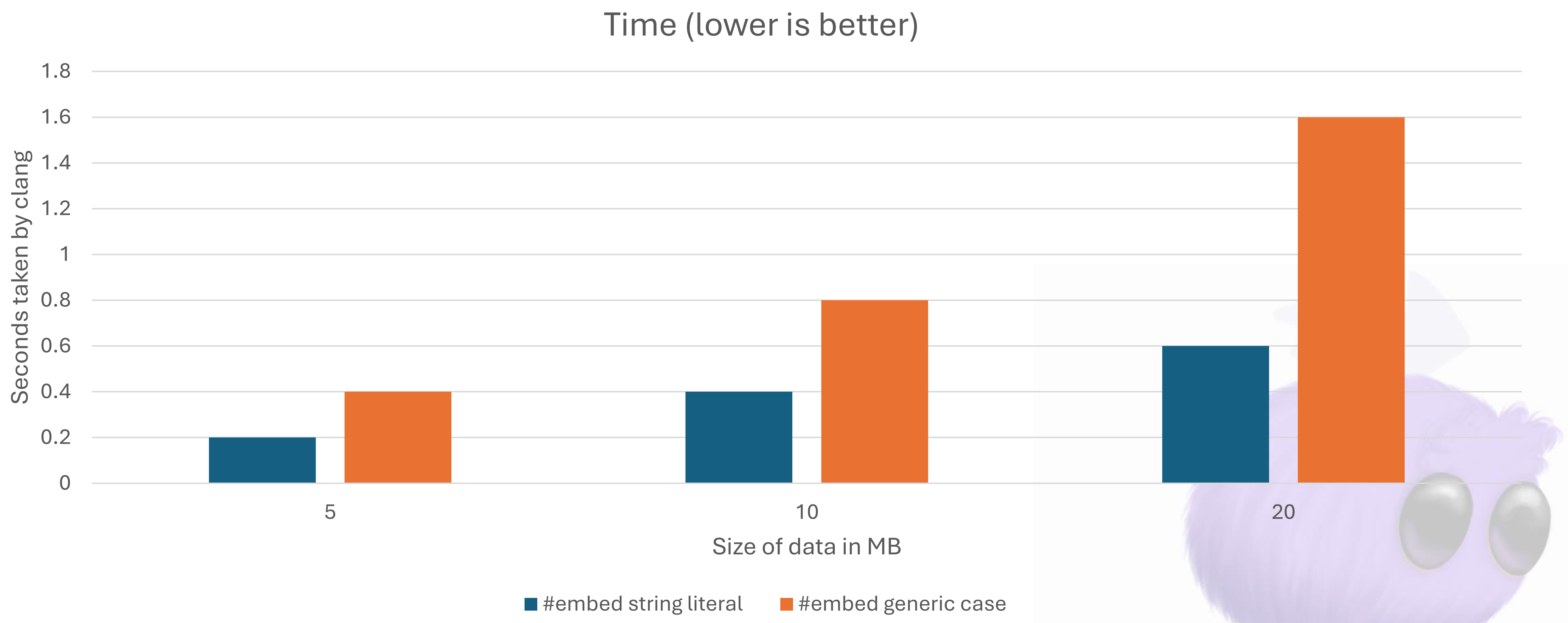
How expensive is that?

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

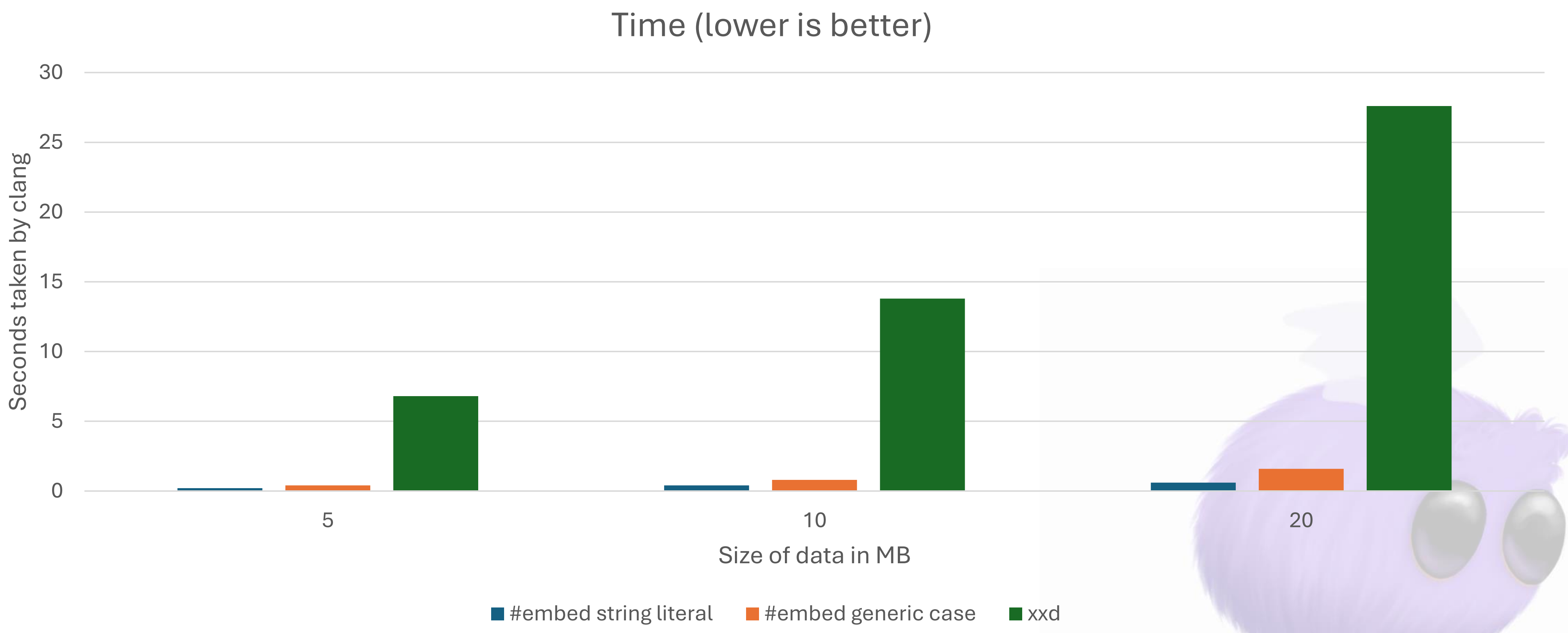
```
// Generic case
int c[] = {1,
#embed "file.bin"
};
```

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

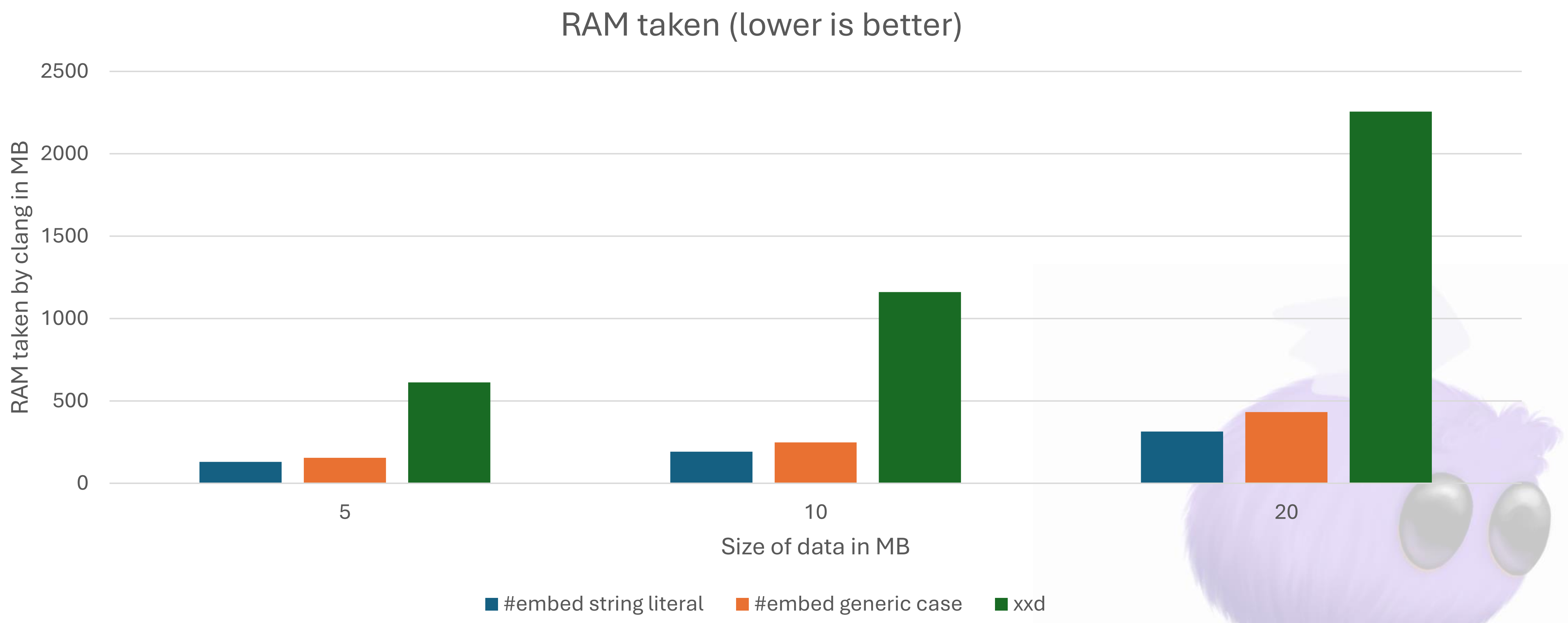
Time difference



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

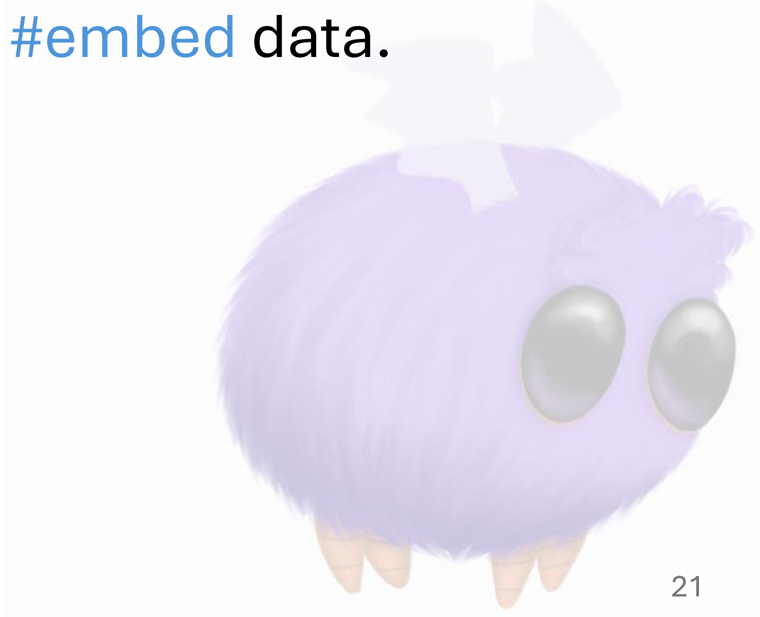
#embed annotation token

```
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.