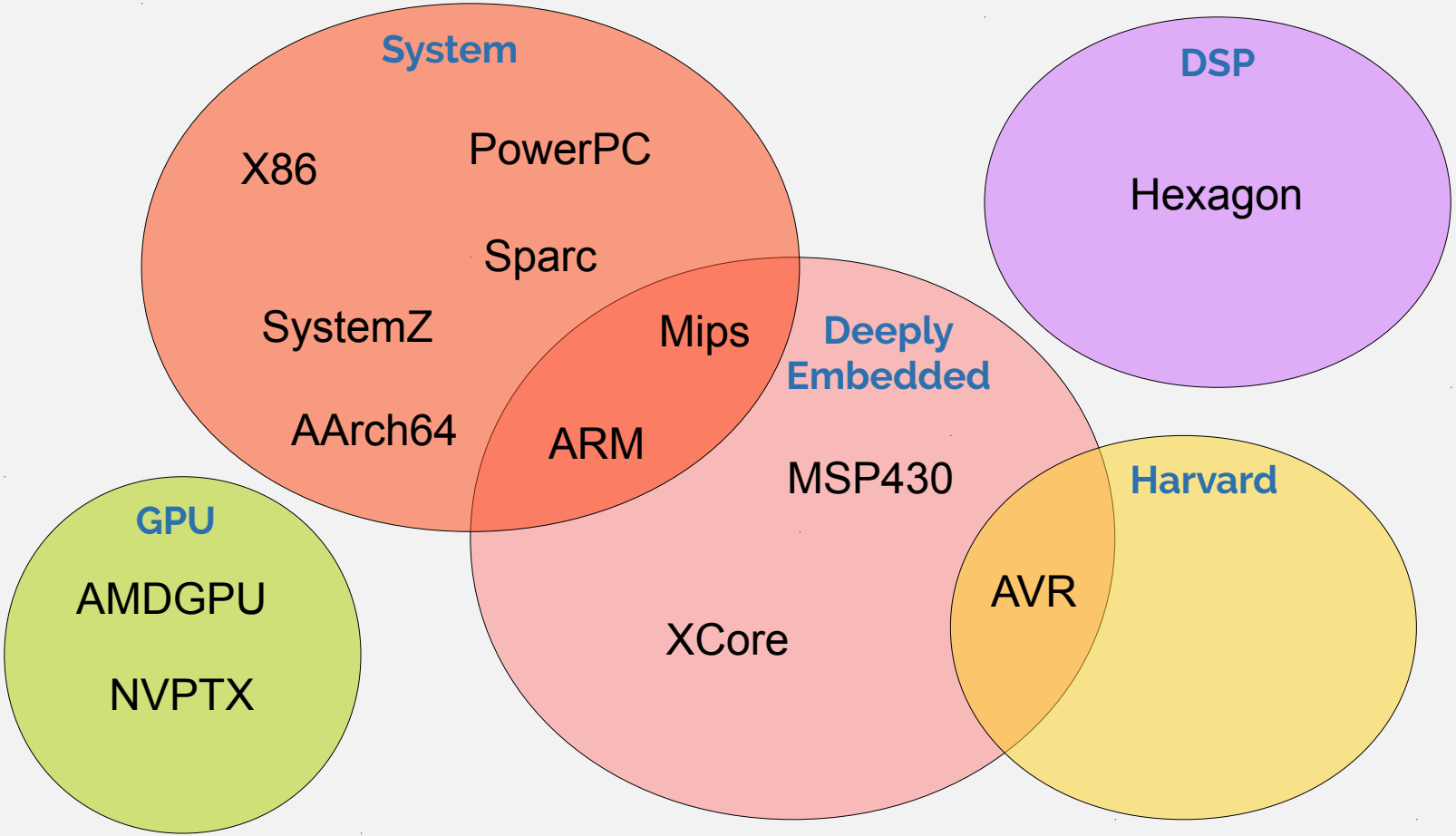
An Altruistic Processor

Ed Jones

EMBECOSM®

Current in-tree architectures



Motivation

- Lots of out of tree targets too!
- Features often not in LLVM
- Hard to submit patches
  - No comparable in tree backend
  - Patches tend to bit rot

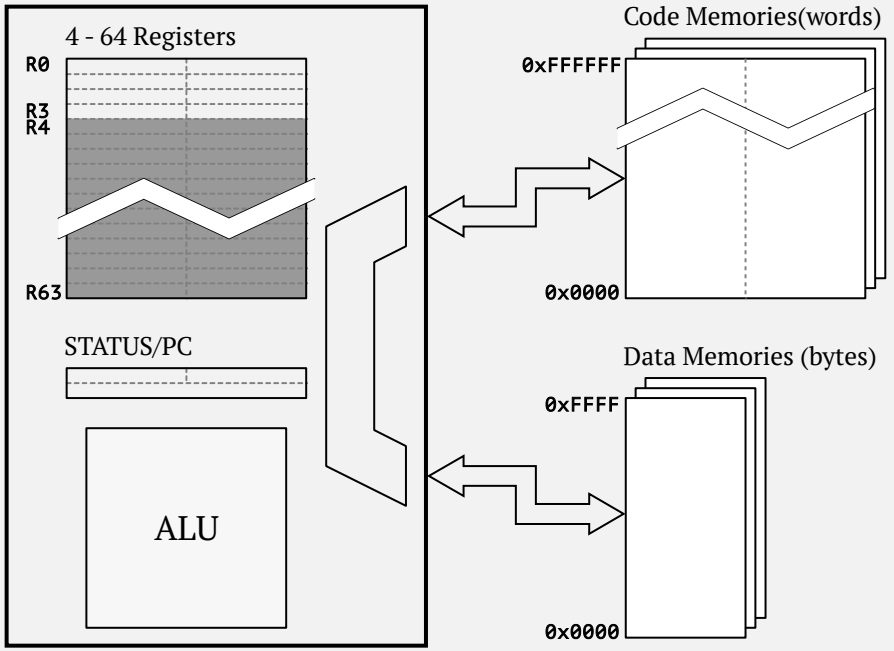
Motivation

- Experimentation platform
  - Add missing features
- Design flexible
- Broaden the appeal of LLVM

Okay enough background, what does AAP look like?

What is AAP?

- Small embedded Harvard architecture
  - 64kB of Byte addressed data
  - 16MW of Word addressed code
- 3 operand instructions
- 16/32/48 bit encoding
- Load store architecture
- Configurable register count (4 – 64)

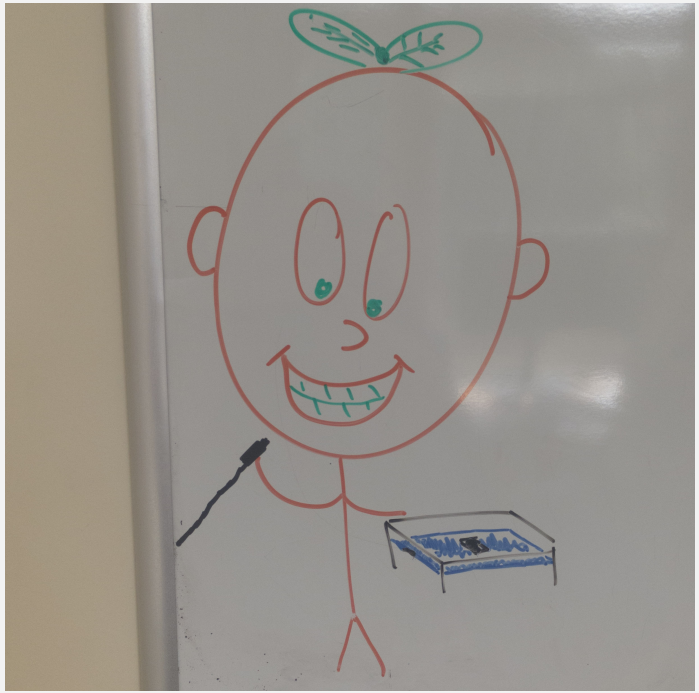


What is AAP?

```
    .globl   main
    .align   4
    .type main,@function
main:                                   ; @main
; BB#0:                                 ; %entry
    subi  $r1, $r1, 2
    stw [$r1, 0], $r0             ; 2-byte Folded Spill
    movi  $r2, .L.str
    bal printString, $r0
    movi  $r2, 0
    ldw $r0, [$r1, 0]             ; 2-byte Folded Reload
    addi  $r1, $r1, 2
    jmp $r0
.Lfunc_end1:
    .size main, .Lfunc_end1-main
```

AAP: Status

- Clang and LLVM implementation
- GNU ld and GDB port
- Initial FPGA implementation by Dan Gorringe
  - Presented at ORCONF at CERN last year
- ISA version 2.1 published
  - www.embecosm.com/EAN13
- AAPSim, using LLVM MC layer
  - Simon Cook will talk about this next
- Andrew Burgess has begun work on a GCC port



Now lets look in detail at some problems we're addressing

Some LLVM problems

- Small register counts
- Postinc, predec addressing modes
- sizeof(int (fnptr*)()) > sizeof(void *)
- When your stack is cheaper than registers
- Multiple CALL and accompanying RET instructions
- Function pointer size depending on call convention

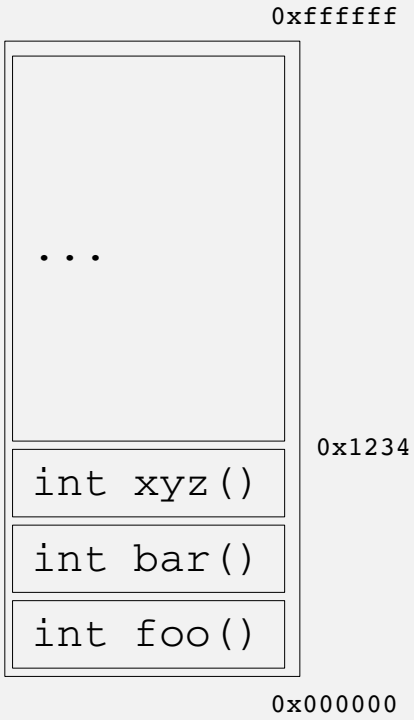
Some LLVM problems

- Small register counts
- Postinc, predec addressing modes
- **sizeof(int (fnptr*)()) > sizeof(void *)**
- **When your stack is cheaper than registers**
- Multiple CALL and accompanying RET instructions
- Function pointer size depending on call convention

sizeof(int (*fnptr)()) > sizeof(void*)

- Code pointers are the same size as data pointers in LLVM
- Our data pointers are 16 bits
  - We need at least 24 bits to address all of our code
- Bad solution: Round up, make data and code pointers 32 bits
  - Must handle 32 bit pointers everywhere in backend
- Better solution: Indirection!
  - Use 16 bit pointers for both, and apply the magic of indirection

sizeof(int (*fnptr)()) > sizeof(void*)

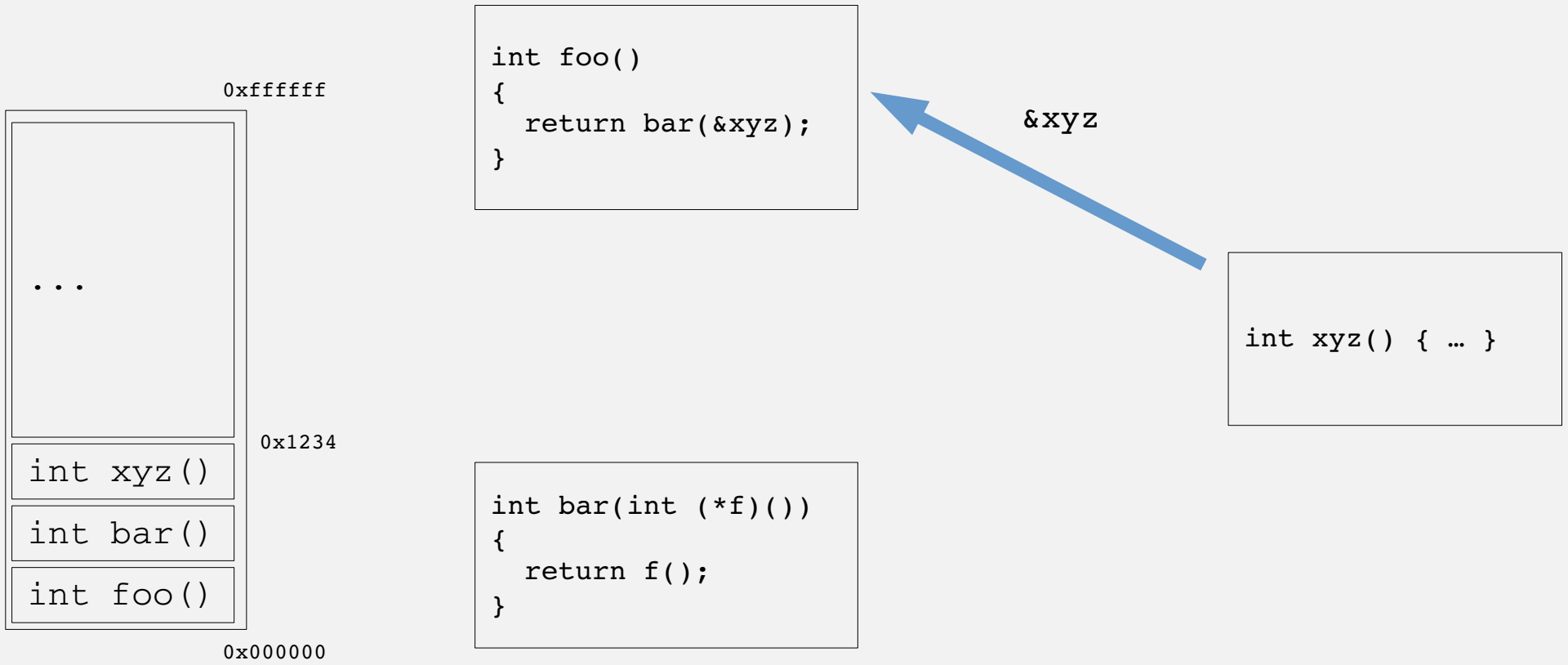



```
int foo()
{
  return bar(&xyz);
}
```

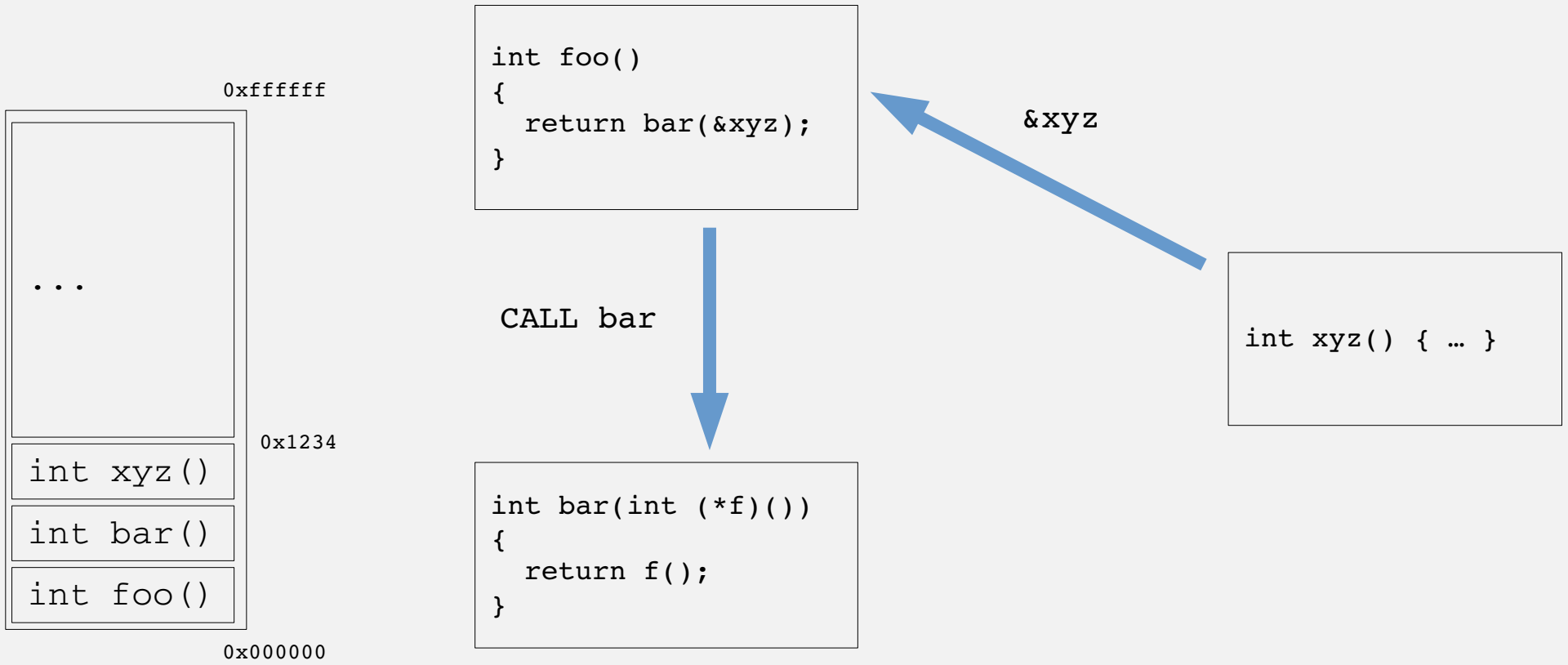
```
int bar(int (*f)())
{
  return f();
}
```

```
int xyz() { … }
```

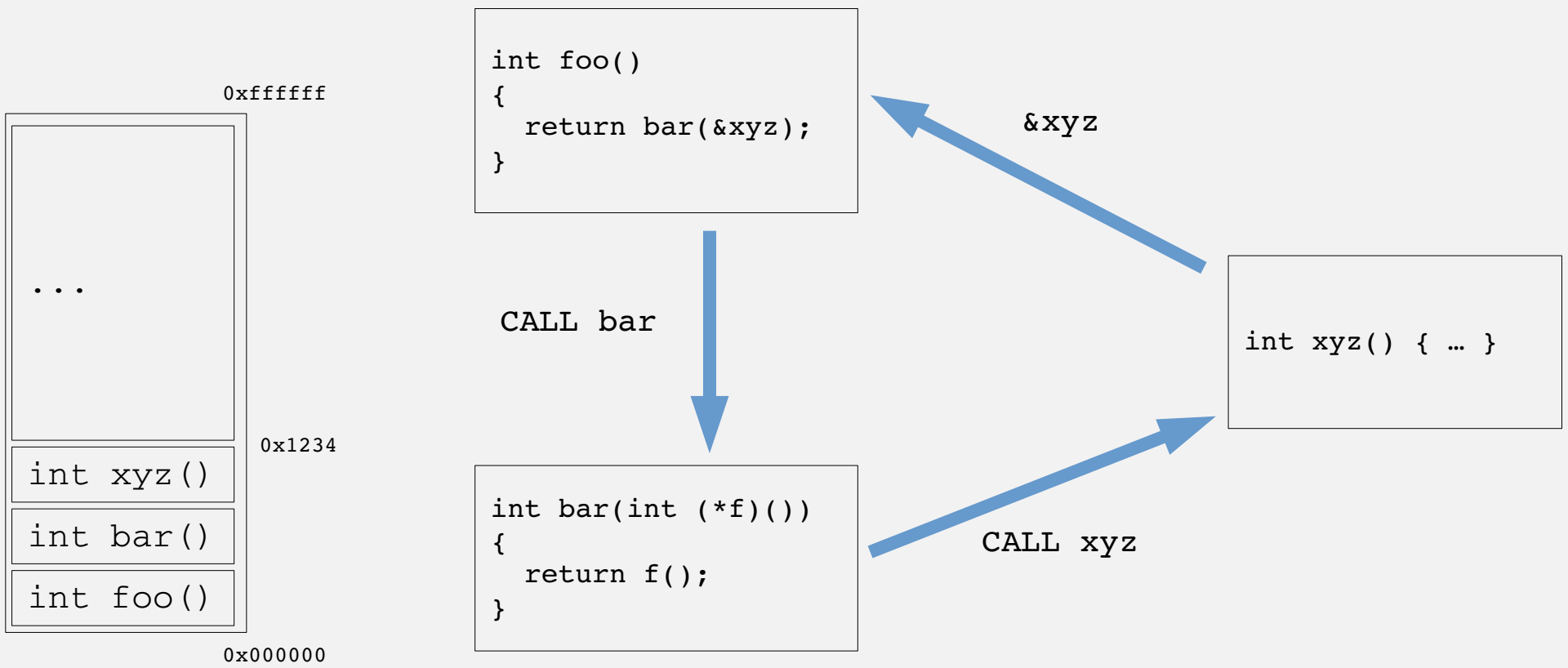
sizeof(int (*fnptr)()) > sizeof(void*)



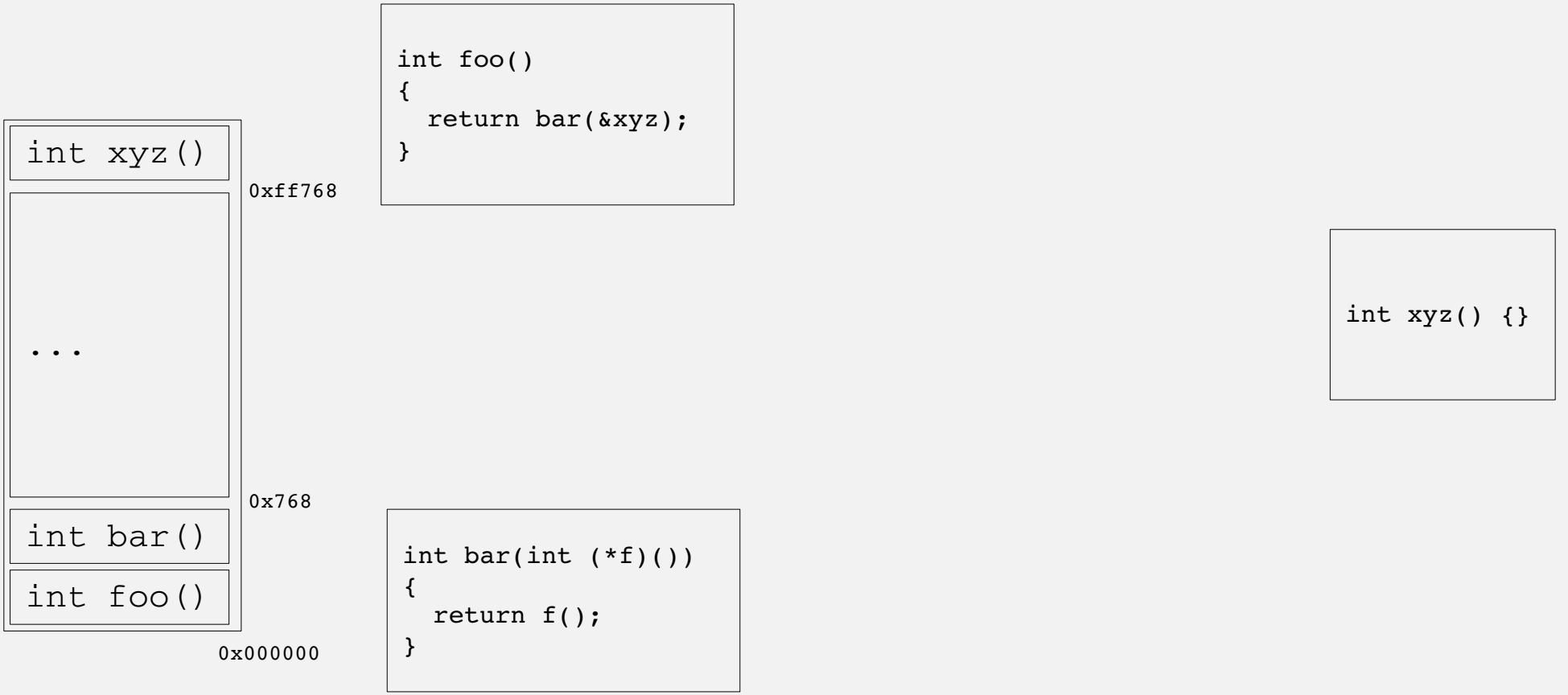
sizeof(int (*fnptr)()) > sizeof(void*)



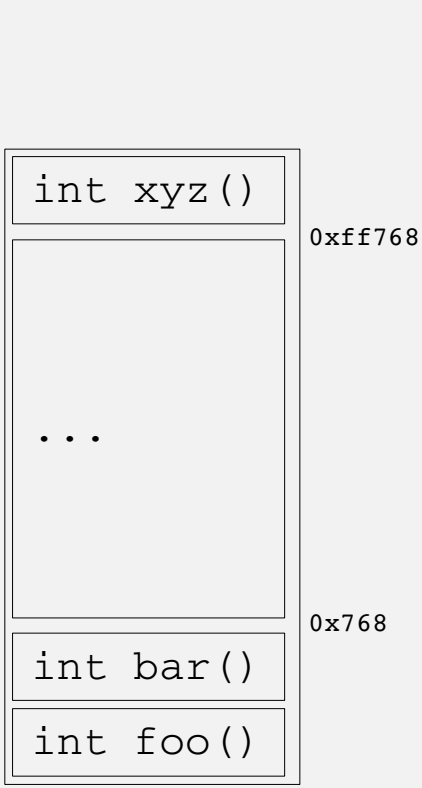
sizeof(int (*fnptr)()) > sizeof(void*)



sizeof(int (*fnptr)()) > sizeof(void*)



sizeof(int (*fnptr)()) > sizeof(void*)



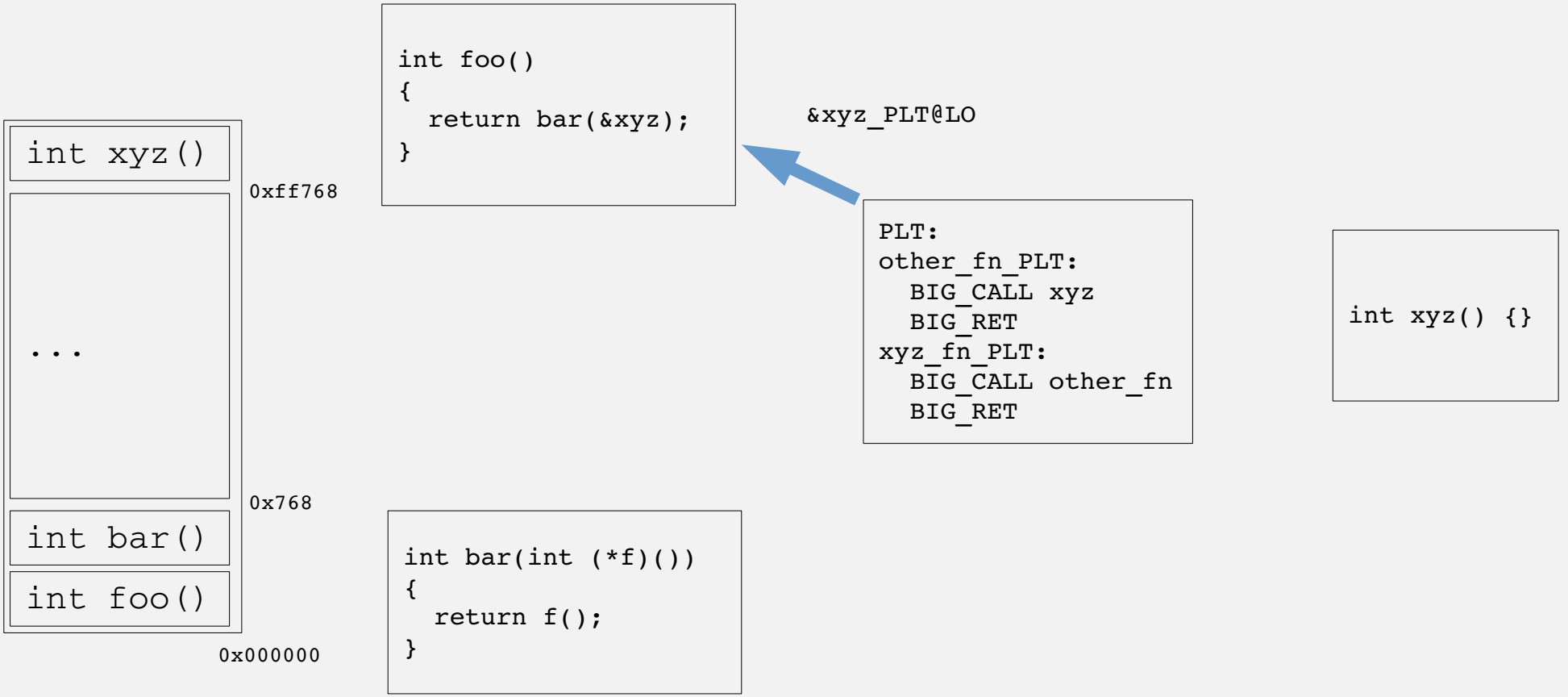

```
int foo()
{
  return bar(&xyz);
}
```

```
PLT:
other_fn_PLT:
  BIG_CALL xyz
  BIG_RET
xyz_fn_PLT:
  BIG_CALL other_fn
  BIG_RET
```

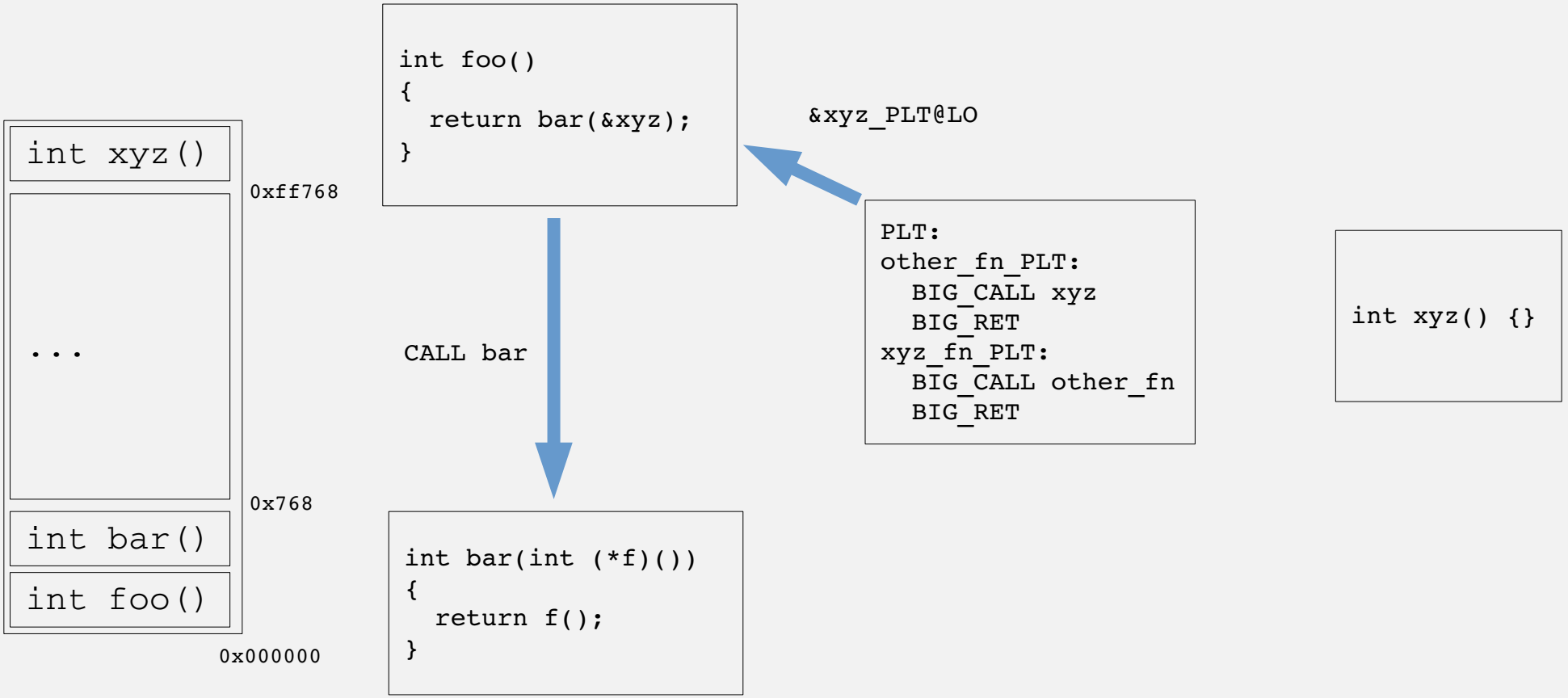
```
int xyz() {}
```

```
int bar(int (*f)())
{
  return f();
}
```

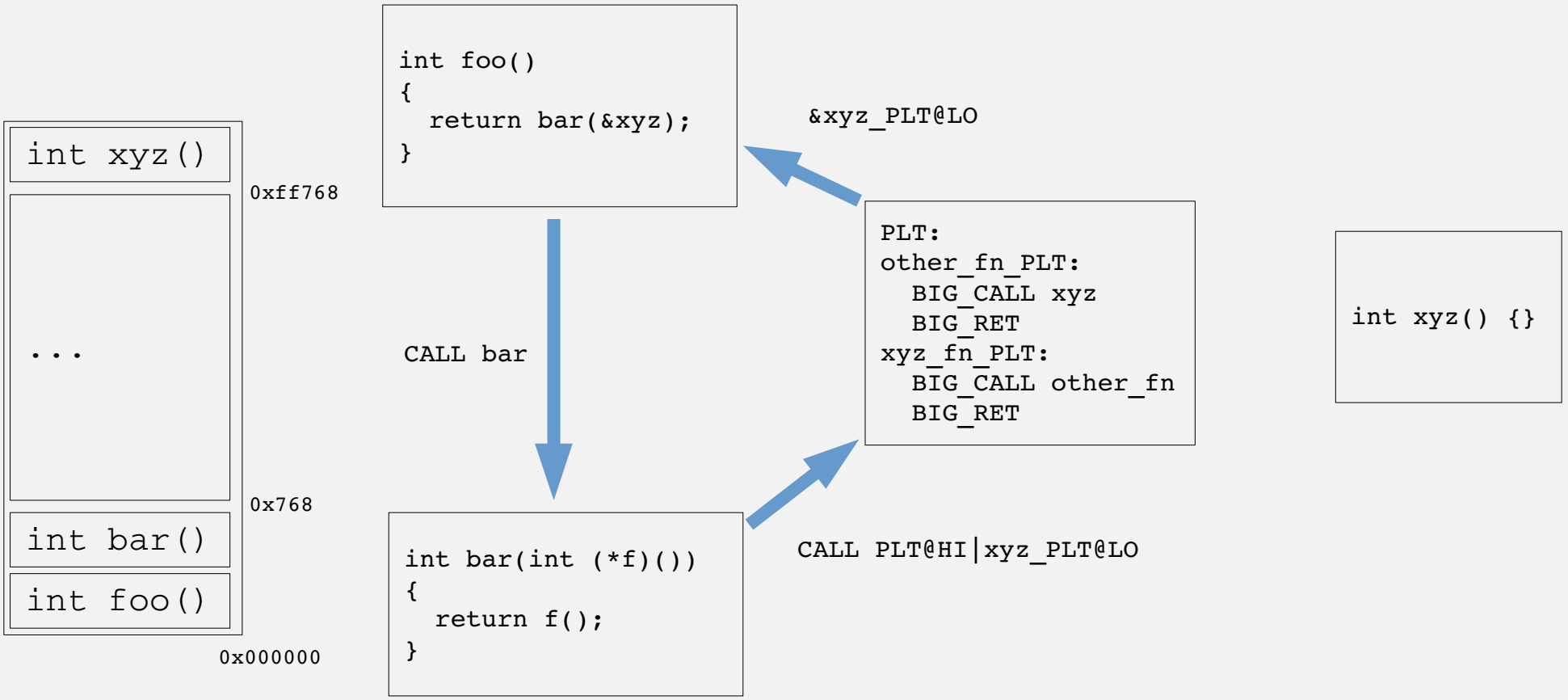
sizeof(int (*fnptr)()) > sizeof(void*)



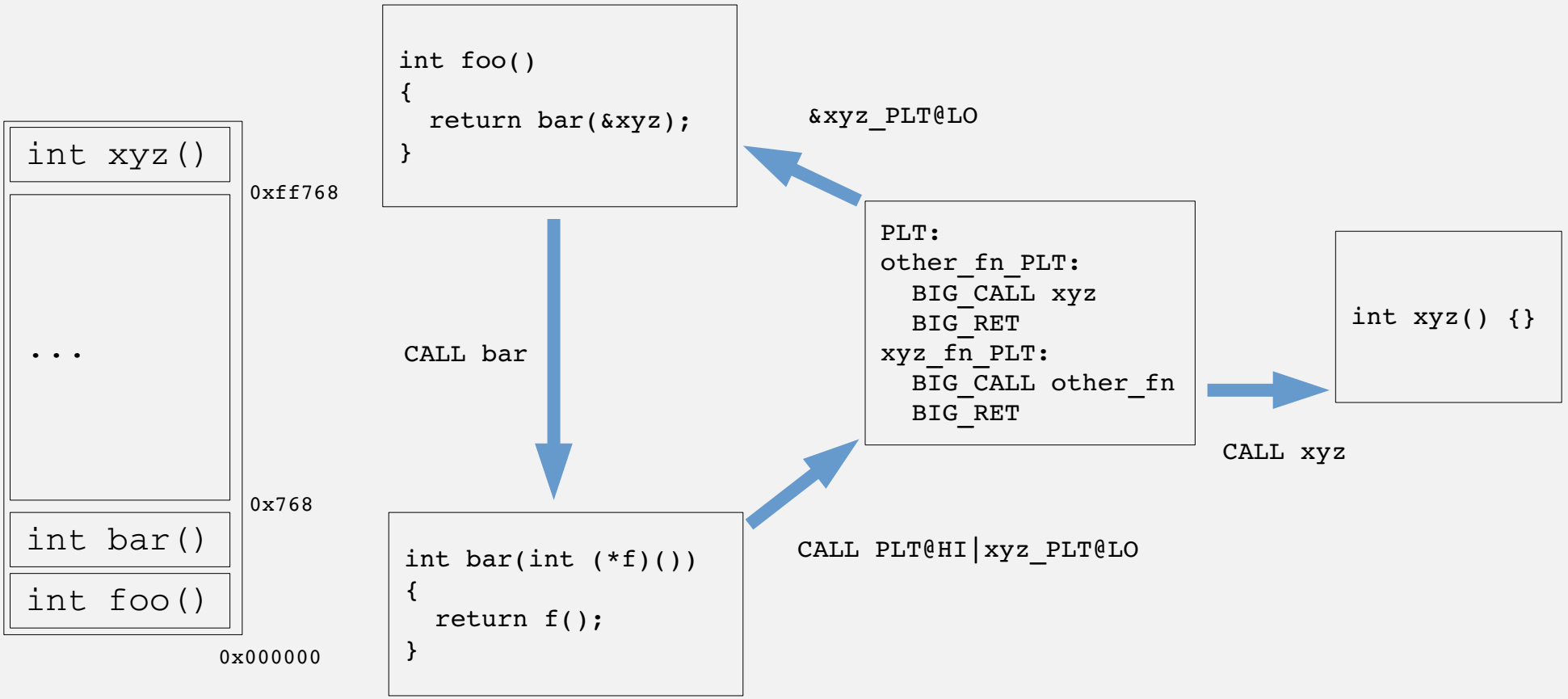
sizeof(int (*fnptr)()) > sizeof(void*)



sizeof(int (*fnptr)()) > sizeof(void*)

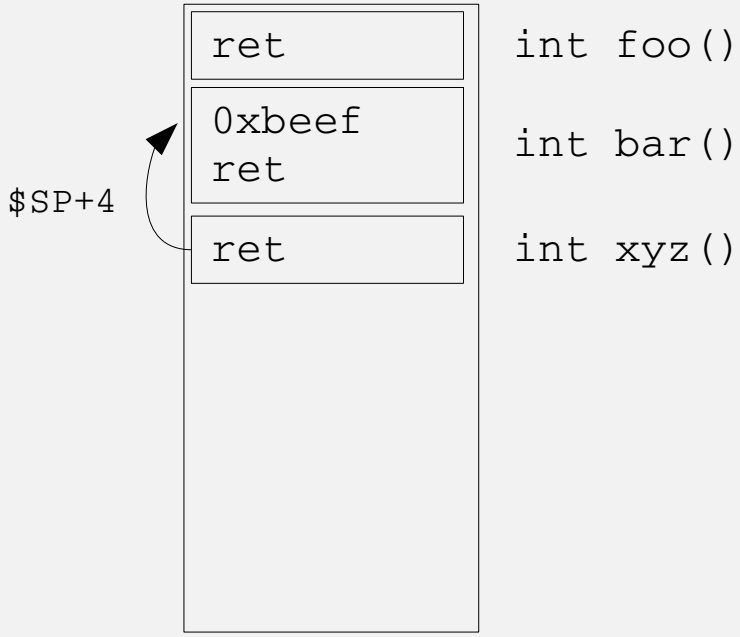


sizeof(int (*fnptr)()) > sizeof(void*)



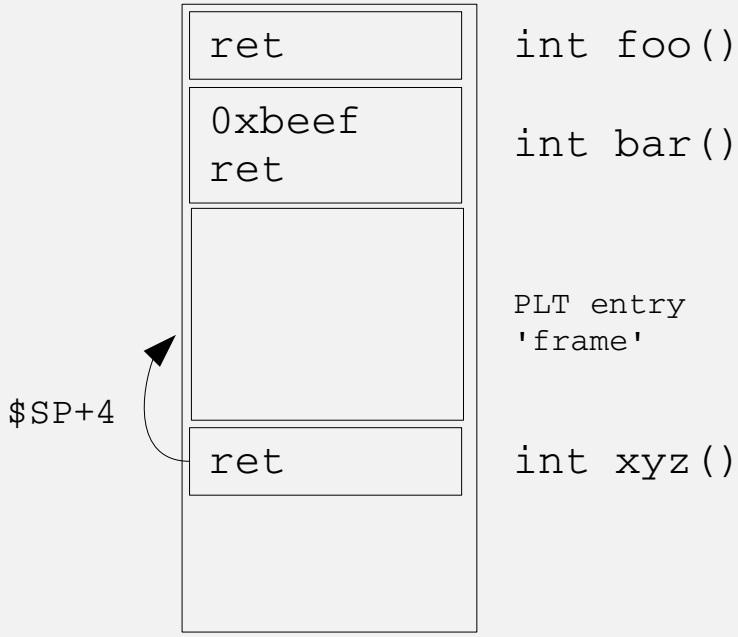
sizeof(int (*fnptr)()) > sizeof(void*)

- Indirection works, but…
- Costly both in size and performance
- Constraints on PLT entry
  - Must be transparent to the callee
  - Might not be possible



sizeof(int (*fnptr)()) > sizeof(void*)

- Indirection works, but…
- Costly both in size and performance
- Constraints on PLT entry
  - Must be transparent to the callee
  - Might not be possible



sizeof(int (*fnptr)()) > sizeof(void*)

- Best solution: Fix the assumption in LLVM that code and data pointers are the same size
- Okay, onto the next problem…

When your stack is cheaper than registers

- What do you mean cheap?
  - ALU operations done directly on some stack slots
  - Avoids load and store into register
- May only be a subset of operations
  - Poses an interesting instruction selection task
- LLVM assumes registers are always cheaper than stack
  - A safe assumption, but architectures aren't always that boring

When your stack is cheaper than registers

- Solution: Pretend some stack slots are registers
  - regalloc is our weapon of choice.
- Where do you spill a stack slot?
  - Another stack slot!
  - May need to load it into a register first

    … this could be a problem
- These “pseudo” registers can't be used as stack
- Okay, this solution is insane…

When your stack is cheaper than registers

- Actual solution: Teach LLVM how to handle this case
- Associate cost with stack slots
- Allow them to be allocated like registers
- Still need to know how to 'spill' stack slots
- Need to describe when these instructions are valid for ISel

State Upstream

- Baseline patches submitted to Phabricator
  - D12191 – LLVM
  - D12192 – Clang
- Reviews and feedback welcome
- The toolchain is also available on our github:

  http://www.github.com/embecosm

Thank You

github.com/embecosm

ed.jones@embecosm.com