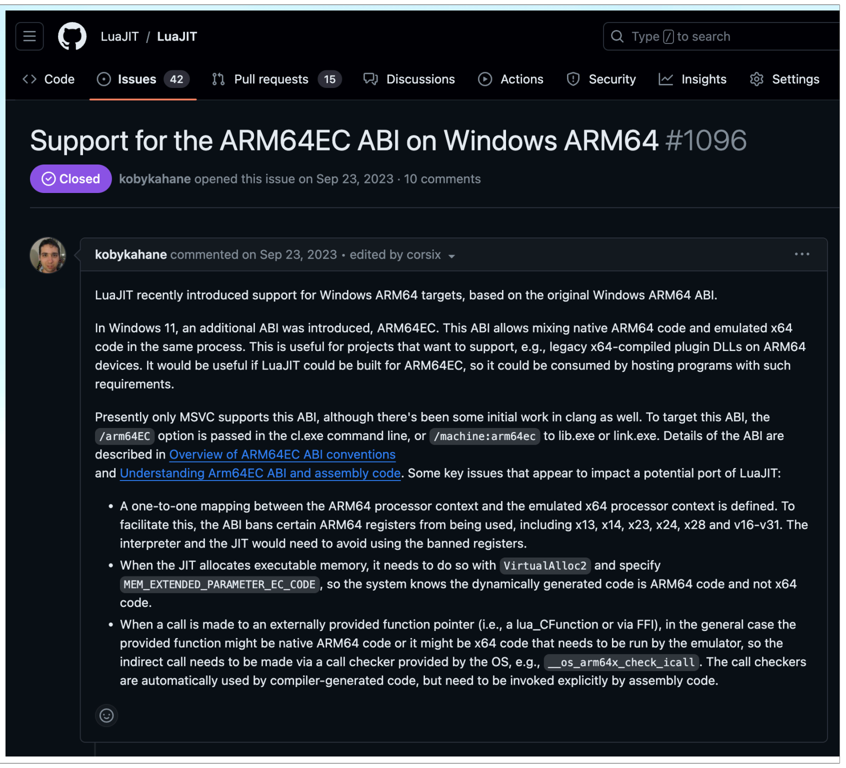
Arm64EC:
Microsoft's Emulation Frankenstein

Peter Cawley (@corsix), 4th February 2024, FOSDEM

Why am I *here*?



Support for the ARM64EC ABI on Windows ARM64 #1096

Closed kobykahane opened this issue on Sep 23, 2023 · 10 comments

kobykahane commented on Sep 23, 2023 • edited by corsix

LuaJIT recently introduced support for Windows ARM64 targets, based on the original Windows ARM64 ABI.

In Windows 11, an additional ABI was introduced, ARM64EC. This ABI allows mixing native ARM64 code and emulated x64 code in the same process. This is useful for projects that want to support, e.g., legacy x64-compiled plugin DLLs on ARM64 devices. It would be useful if LuaJIT could be built for ARM64EC, so it could be consumed by hosting programs with such requirements.

Presently only MSVC supports this ABI, although there's been some initial work in clang as well. To target this ABI, the `/arm64EC` option is passed in the cl.exe command line, or `/machine:arm64ec` to lib.exe or link.exe. Details of the ABI are described in Overview of ARM64EC ABI conventions
and Understanding Arm64EC ABI and assembly code. Some key issues that appear to impact a potential port of LuaJIT:

- A one-to-one mapping between the ARM64 processor context and the emulated x64 processor context is defined. To facilitate this, the ABI bans certain ARM64 registers from being used, including x13, x14, x23, x24, x28 and v16-v31. The interpreter and the JIT would need to avoid using the banned registers.
- When the JIT allocates executable memory, it needs to do so with `VirtualAlloc2` and specify `MEM_EXTENDED_PARAMETER_EC_CODE`, so the system knows the dynamically generated code is ARM64 code and not x64 code.
- When a call is made to an externally provided function pointer (i.e., a lua_CFunction or via FFI), in the general case the provided function might be native ARM64 code or it might be x64 code that needs to be run by the emulator, so the indirect call needs to be made via a call checker provided by the OS, e.g., `__os_arm64x_check_icall`. The call checkers are automatically used by compiler-generated code, but need to be invoked explicitly by assembly code.

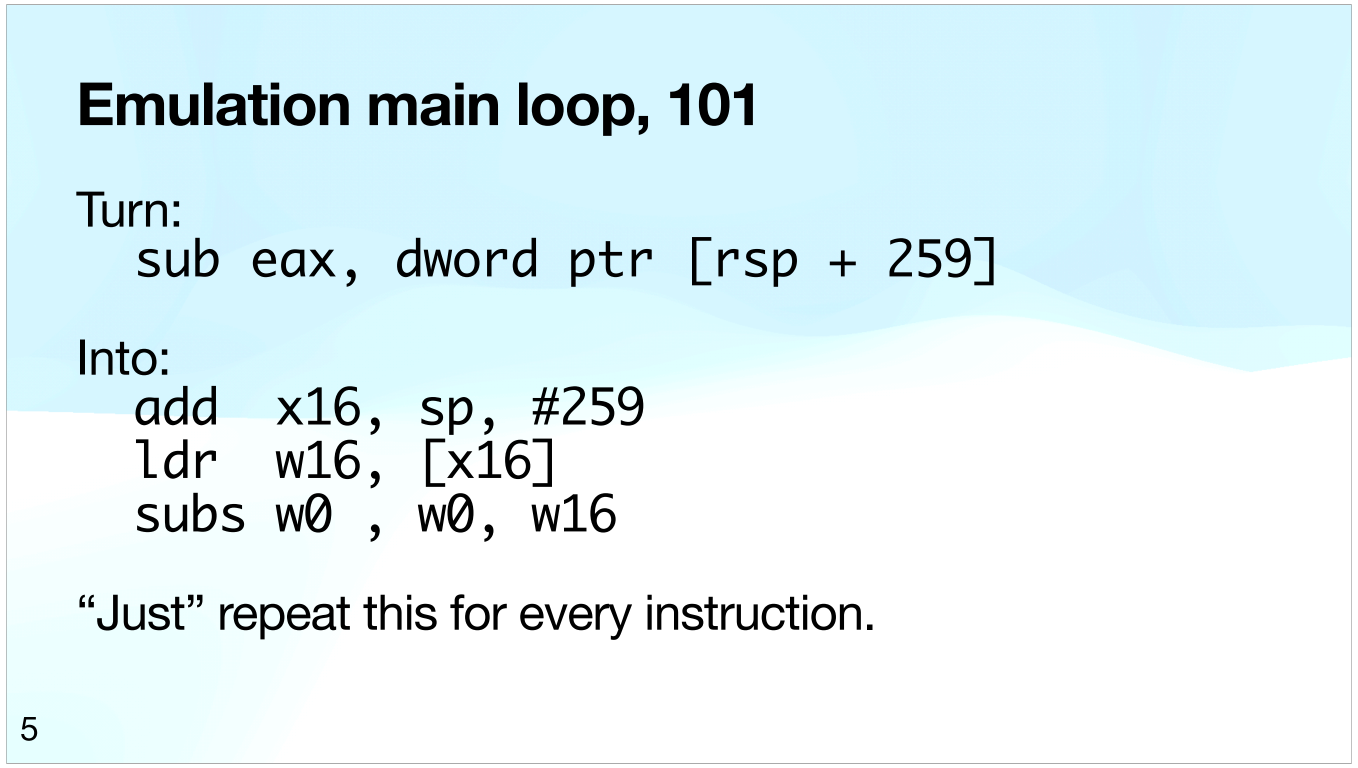
You might be asking why I'm here at FOSDEM talking about a closed-source Microsoft technology.
I hack on LuaJIT as a hobby, which is FOSS, and recently gained support for Windows on ARM64.
Then somebody came along and asked about a _different_ Windows on ARM64.
I was intrigued by the Frankenstein they described, and thought you might be too.

Disclaimers

- I do not work for Microsoft.
- I do not work for Intel.
- I do not work for Arm.
- I am not Mike Pall.
- Views are my own.

Agenda

1. General landscape of emulating x64 on arm64. ←
2. What is Arm64EC?
3. Lessons learnt porting LuaJIT to Arm64EC.



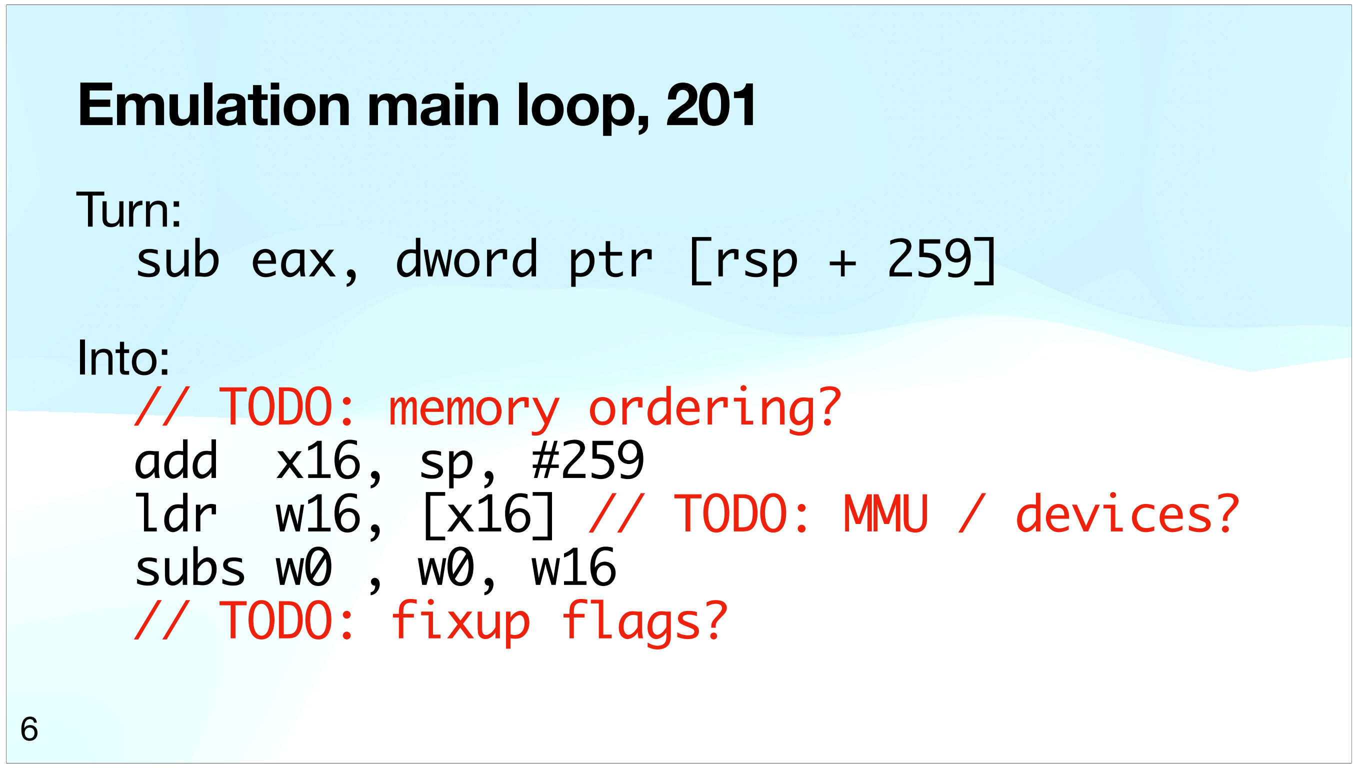
Emulation main loop, 101

Turn:
```
  sub eax, dword ptr [rsp + 259]
```

Into:
```
  add  x16, sp, #259
  ldr  w16, [x16]
  subs w0 , w0, w16
```

“Just” repeat this for every instruction.

Emulating x64 on arm64 initially looks simple; just convert every x64 instruction into an equivalent sequence of arm64 instructions, and feed them to an arm64 CPU. One instruction might become several, but such is CISC versus RISC.



Emulation main loop, 201

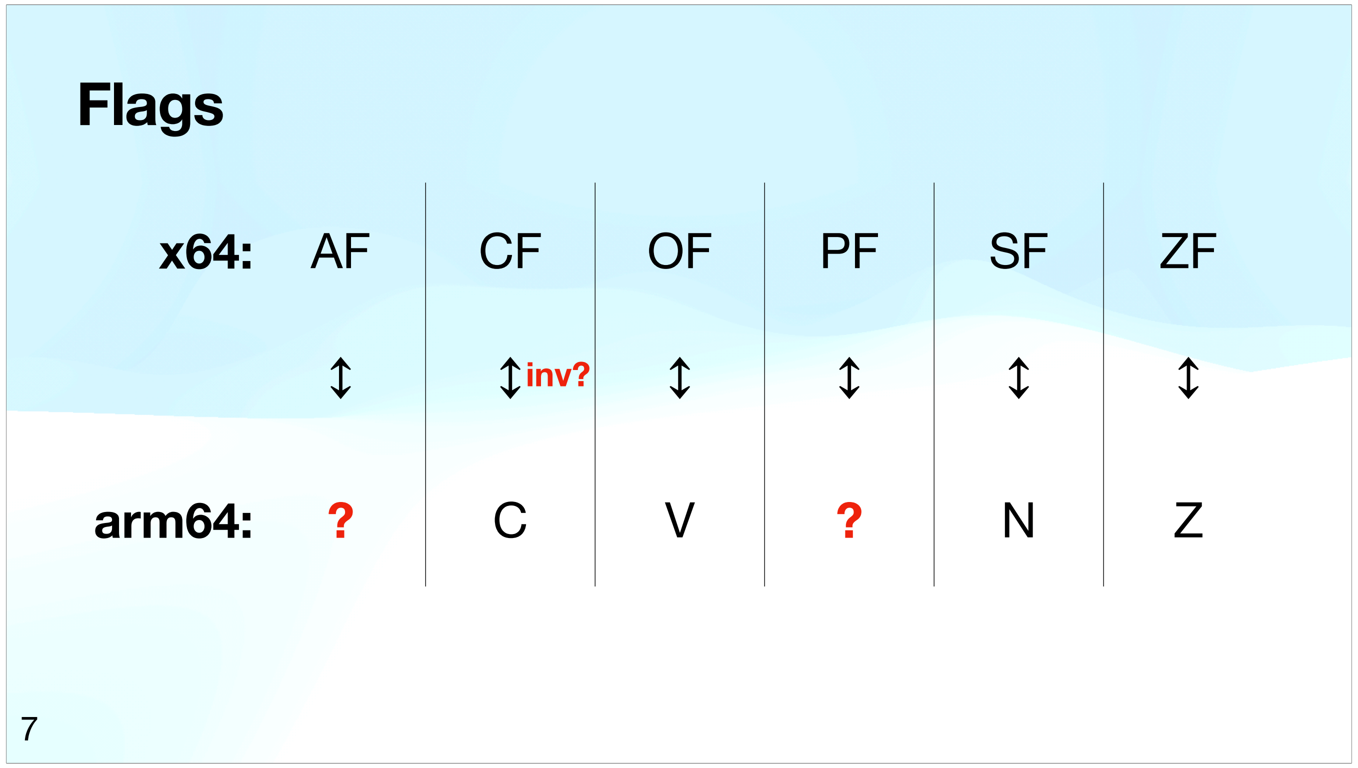
Turn:

```
  sub eax, dword ptr [rsp + 259]
```

Into:

```
  // TODO: memory ordering?
  add  x16, sp, #259
  ldr  w16, [x16] // TODO: MMU / devices?
  subs w0 , w0, w16
  // TODO: fixup flags?
```

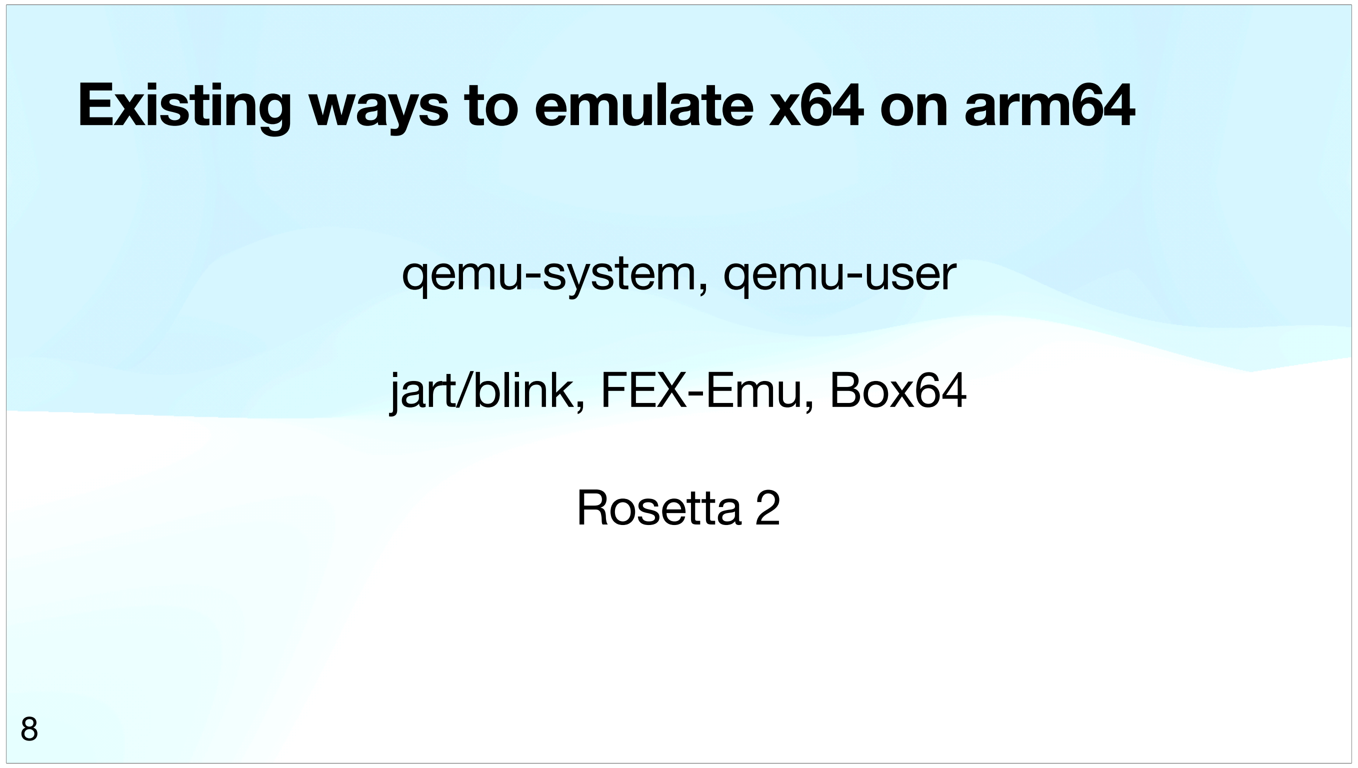
But oh boy is the devil in the details. Firstly there's memory ordering; x64 has stronger memory order guarantees than arm64. Most instructions are fine with weaker guarantees, but the emulator doesn't know which, so has to issue memory barriers for every load and store. Then there's whether to emulate an MMU, or just defer to the host MMU. Finally there's flags.



Flags

x64:	AF	CF	OF	PF	SF	ZF
	↕	↕ inv?	↕	↕	↕	↕
arm64:	?	C	V	?	N	Z

Most x64 instructions produce an array of six flags in addition to their main result. Some arm64 instructions produce an array of four flags in addition to their main result. Two problems; some versus most, and four versus six. The missing two are AF and PF, neither of which are actually needed by most x64 code, but it can be hard for the emulator to know when they're not needed. PF gives parity of low 8 bits of main result, which is annoying but straightforward to compute. AF gives the carry/borrow out of the low four bits, meant to speed up BCD arithmetic, and is more annoying to compute. CF is interesting because x64 / arm64 choose opposite interpretations for CF in subtraction, so inverting the bit can be necessary; arm64v8.4 added a cfinv instruction just for this.



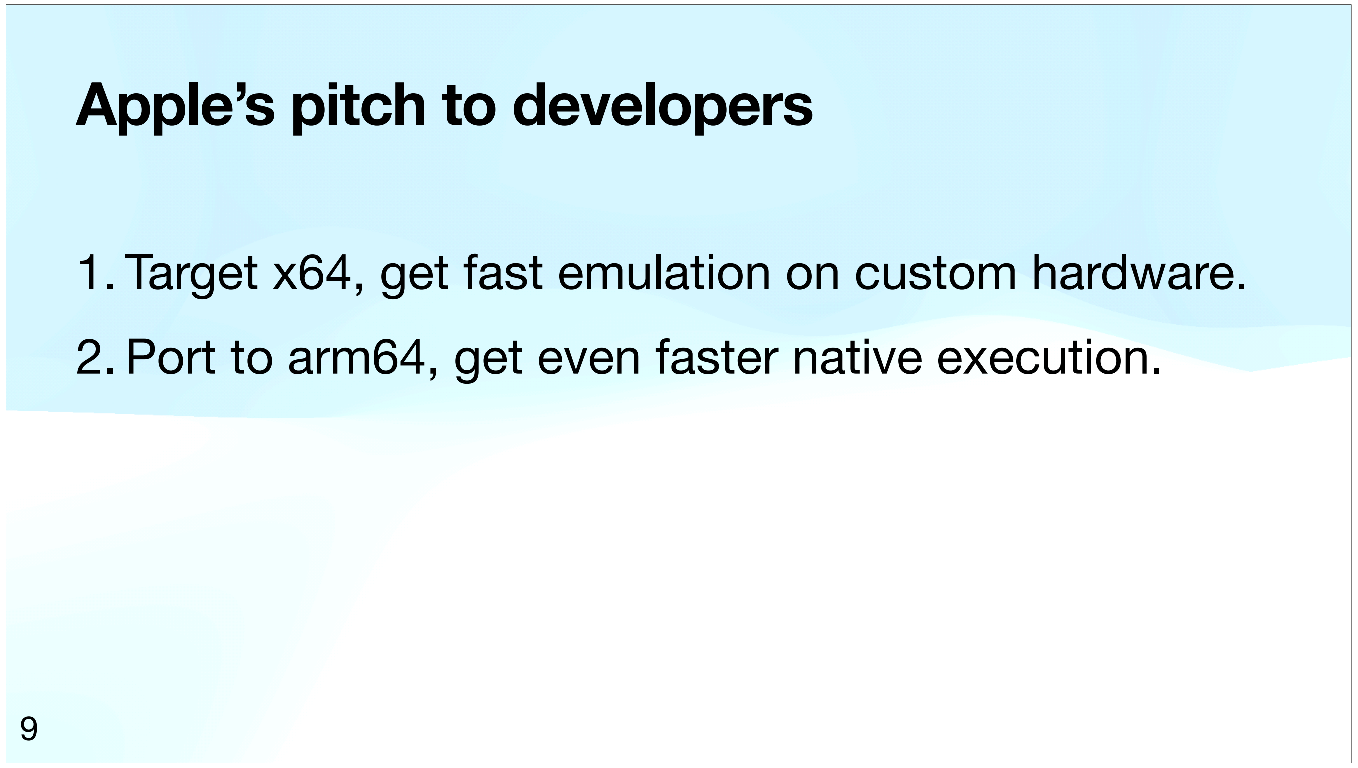
Existing ways to emulate x64 on arm64

qemu-system, qemu-user

jart/blink, FEX-Emu, Box64

Rosetta 2

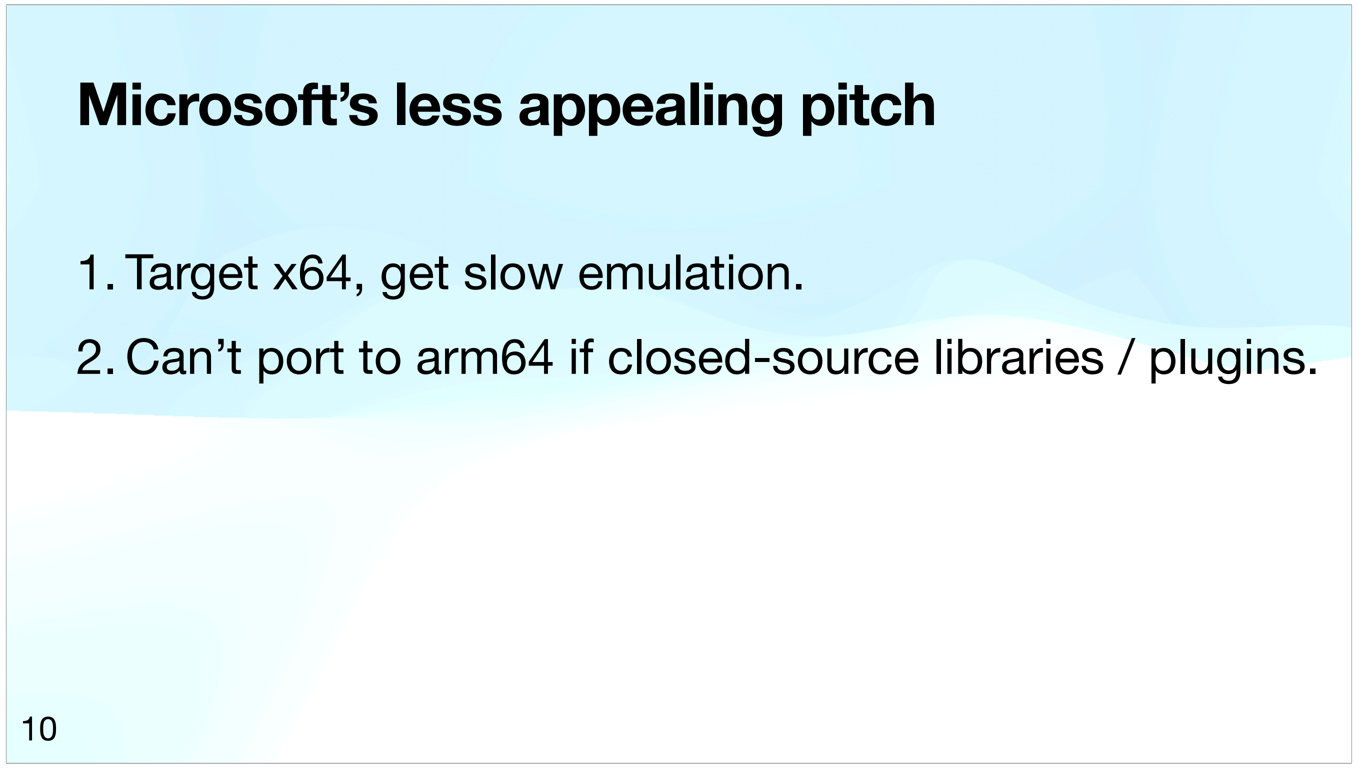
qemu is a FOSS stalwart, emulating anything on anything. qemu-system emulates a bunch of hardware in addition to emulating an x64 CPU, allowing you to run an x64 OS inside. qemu-user emulates an OS in addition to emulating a CPU, allowing you to run an x64 program inside. This can be substantially faster, as the OS logic is native arm64 rather than emulated x64. Then there are a bunch of lesser-known FOSS entrants; Justine Tunney's blink emulates x64 on any architecture, as part of her actually portable executables project. It is similar to qemu-user, though can optionally emulate an MMU. FEX is, I think, trying to be faster than qemu-user by focusing on a much smaller matrix of supported architectures. Box64 is interesting because it knows how to replace certain shared libraries with native versions, allowing the code therein to run natively rather than emulated. The recent entrant on the scene is Apple's Rosetta 2, which can exploit special hardware in Apple's arm64 chips for stronger memory ordering and computing the missing two flags. It performs extremely well.



Apple's pitch to developers

1. Target x64, get fast emulation on custom hardware.
2. Port to arm64, get even faster native execution.

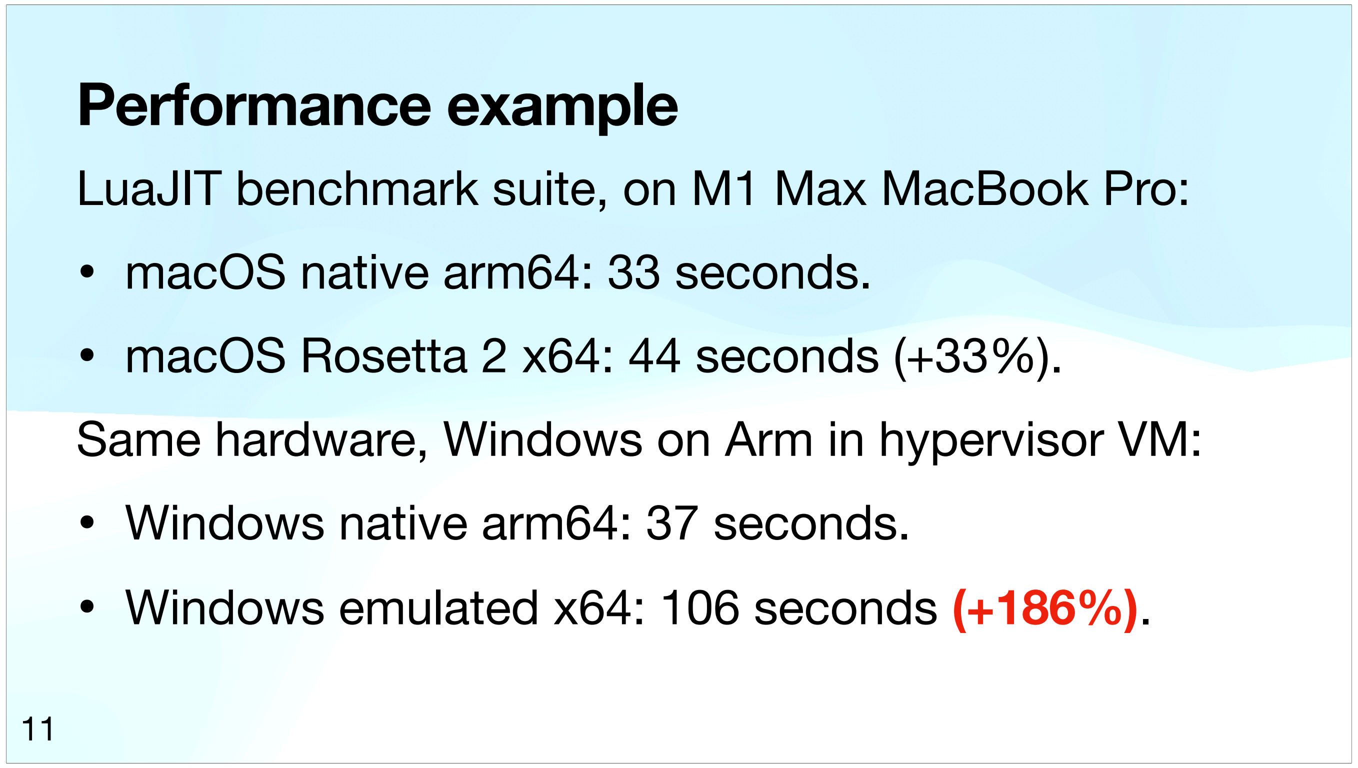
Rosetta 2 means Apple can make a very appealing pitch to developers; keep targeting x64 and you'll get fast emulation, and if that isn't fast enough, you can port to arm64 for even faster native execution. Apple leads the way by porting a bunch of 1st-party software to arm64, and developers in their ecosystem are used to following Apple's lead.



Microsoft’s less appealing pitch

1. Target x64, get slow emulation.
2. Can’t port to arm64 if closed-source libraries / plugins.

Meanwhile, Microsoft are in a much harder place. They can’t rely on the presence of custom hardware, so the x64 emulation will be slow. Developers in their ecosystem often make use of closed-source libraries, or support runtime plugins, both of which make porting to arm64 hard.



Performance example

LuaJIT benchmark suite, on M1 Max MacBook Pro:

- macOS native arm64: 33 seconds.
- macOS Rosetta 2 x64: 44 seconds (+33%).

Same hardware, Windows on Arm in hypervisor VM:

- Windows native arm64: 37 seconds.
- Windows emulated x64: 106 seconds **(+186%)**.

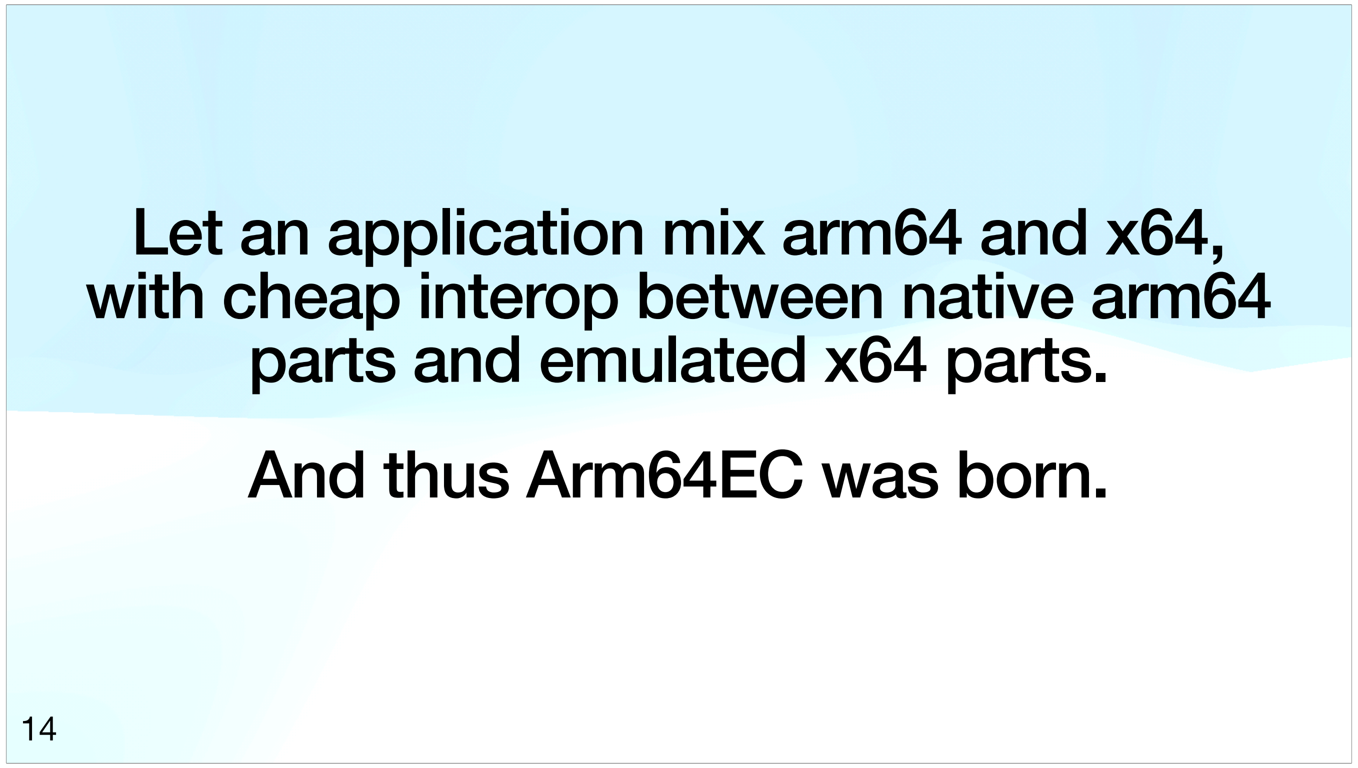
To drive home the performance point, we can look at LuaJIT's benchmark suite. It may or may not be representative of anything else, but it is something. Additionally, a JIT compiler is about the worst thing to run under emulation, as the emulator has to be constantly re-converting JIT-compiled code. The baseline is set by macOS native arm64 at 33 seconds. Rosetta 2 comes in 33% slower, which is really quite good all things considered. Under Windows, arm64 comes in at 37 seconds. I'm not sure how much of the slowdown is due to Windows, how much is due to the hypervisor, and how much is due to 4k versus 16k pages, but 37 seconds is in the same ballpark as 33 seconds. Meanwhile, using Windows x64 emulation results in a catastrophic 106 seconds. Did I mention that Rosetta 2 is really good?

Option 1 is too slow, and option 2 is impossible, but maybe option 1½ is both fast and possible?

So Microsoft are between a rock and a hard place. Option 1, emulation, is too slow, and option 2, porting to arm64, can be impossible. I like to imagine some mad scientist at Microsoft looking at this situation, and thinking: maybe we can blend these two bad options, and end up in a good place?

Agenda

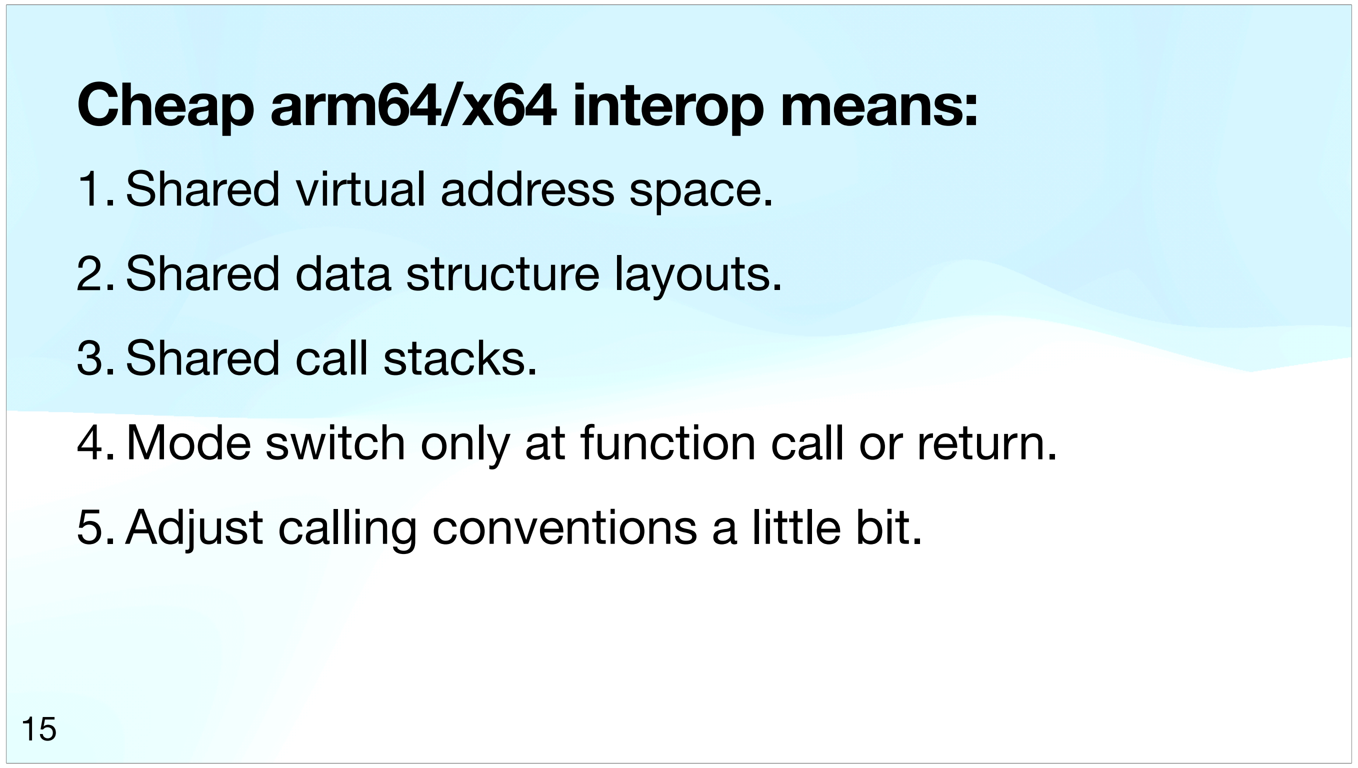
1. General landscape of emulating x64 on arm64.
2. What is Arm64EC? ←
3. Lessons learnt porting LuaJIT to Arm64EC.



Let an application mix arm64 and x64, with cheap interop between native arm64 parts and emulated x64 parts.

And thus Arm64EC was born.

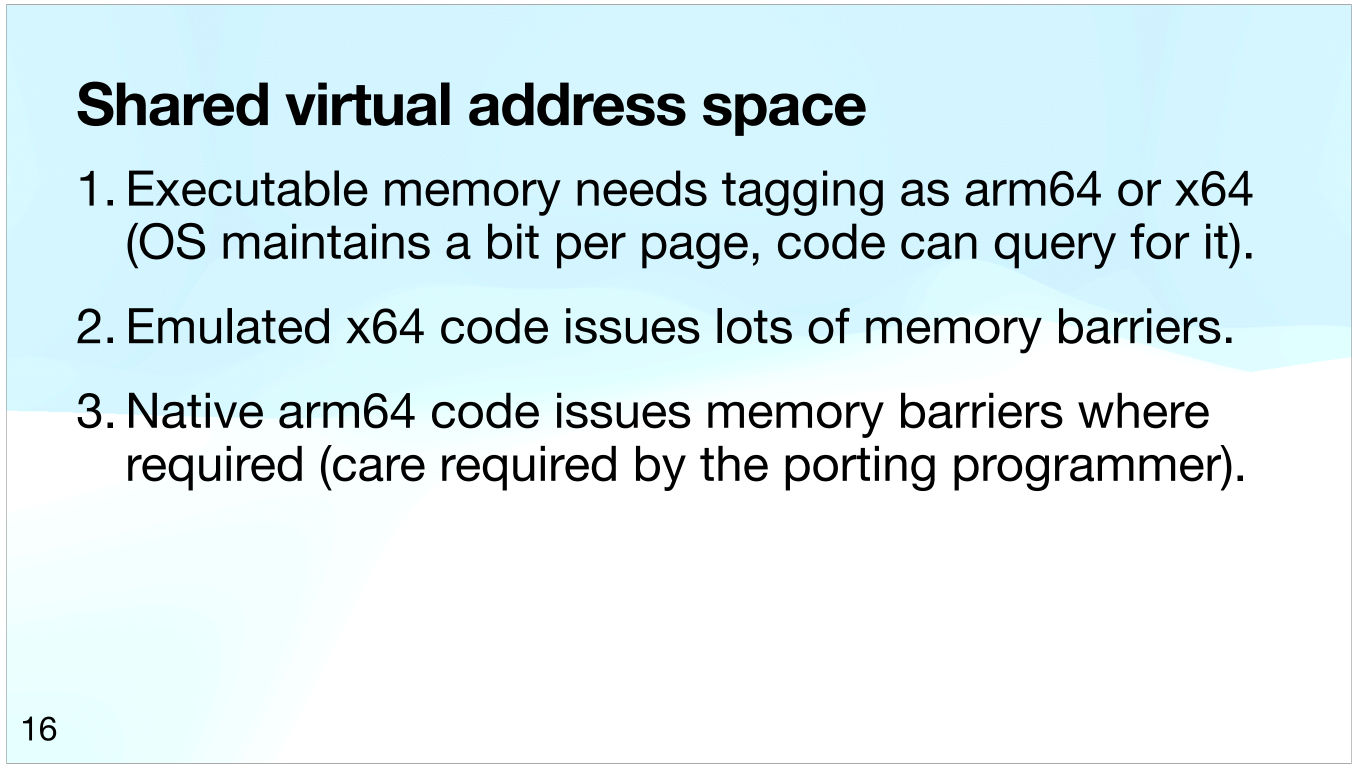
What would blending the options look like? Let an application be a mix of arm64 and x64. Any closed-source libraries or plugins can remain as x64 and be emulated, meanwhile the hot paths of the application can be ported to arm64. More and more of the application can be ported over time, delivering incremental performance improvements. The EC stands for “Emulation Compatible”.



Cheap arm64/x64 interop means:

1. Shared virtual address space.
2. Shared data structure layouts.
3. Shared call stacks.
4. Mode switch only at function call or return.
5. Adjust calling conventions a little bit.

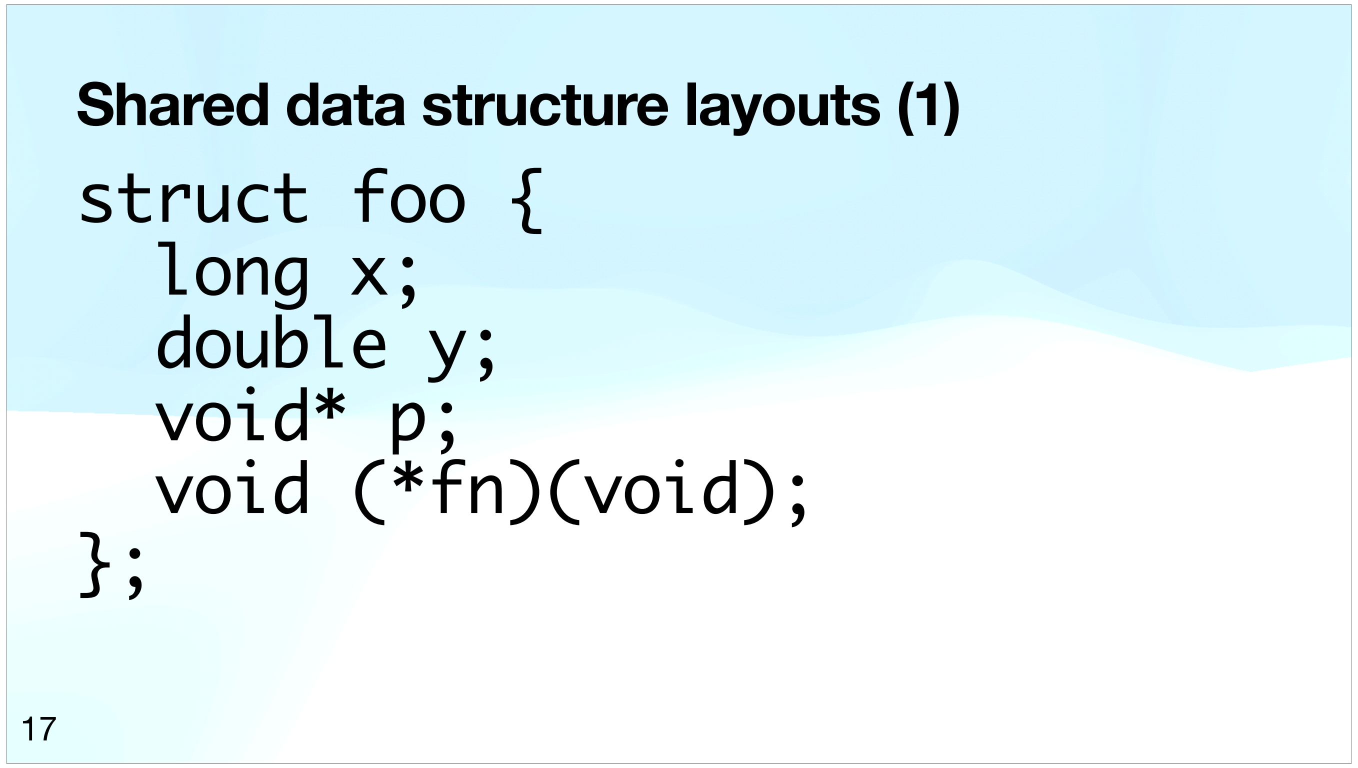
It’s an intriguing idea, but what would it actually mean in practice? Various things need sharing, which we’ll get to. Limiting mode switches to only happen at function call or return makes things simpler than allowing a mode switch at any instruction. In practice you can also mode switch between a throw and a catch, which we’ll touch upon in a bit. Function calling conventions need some minor adjustments, but not too much.



Shared virtual address space

1. Executable memory needs tagging as arm64 or x64 (OS maintains a bit per page, code can query for it).
2. Emulated x64 code issues lots of memory barriers.
3. Native arm64 code issues memory barriers where required (care required by the porting programmer).

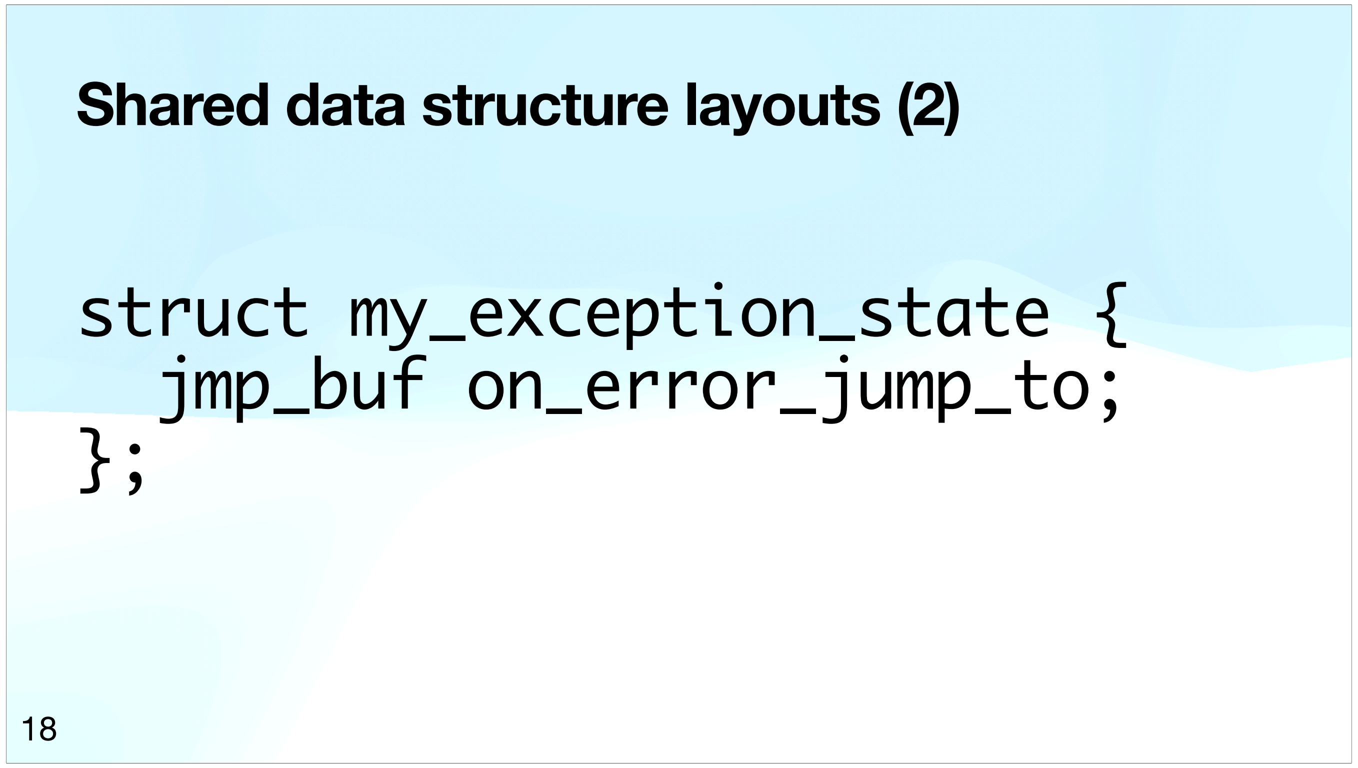
Shared virtual address space means that executable pages within the address space might contain arm64 code, or might contain x64 code, so the OS needs to maintain an extra bit per page specifying the architecture of the code therein. Multiple threads can share an address space, so we need to think about memory ordering. Emulation of the x64 code will involve lots of memory barriers. Meanwhile, the porting programmer is responsible for inserting memory barriers into code ported to arm64.



Shared data structure layouts (1)

```
struct foo {
  long x;
  double y;
  void* p;
  void (*fn)(void);
};
```

Sharing of data structures looks fairly simple to start with. Just make sure that the Arm64EC size/alignment of all types matches that of x64. Long needs to be 4 bytes (this is Windows after all), double at 8 bytes, data pointers at 8 bytes, and function pointers at 8 bytes. That last one is interesting, and is the reason why the OS needs to track the architecture of every executable page: we might prefer to put that information in the function pointer itself, but that would require changing the size of function pointers, which we can't do.



Shared data structure layouts (2)

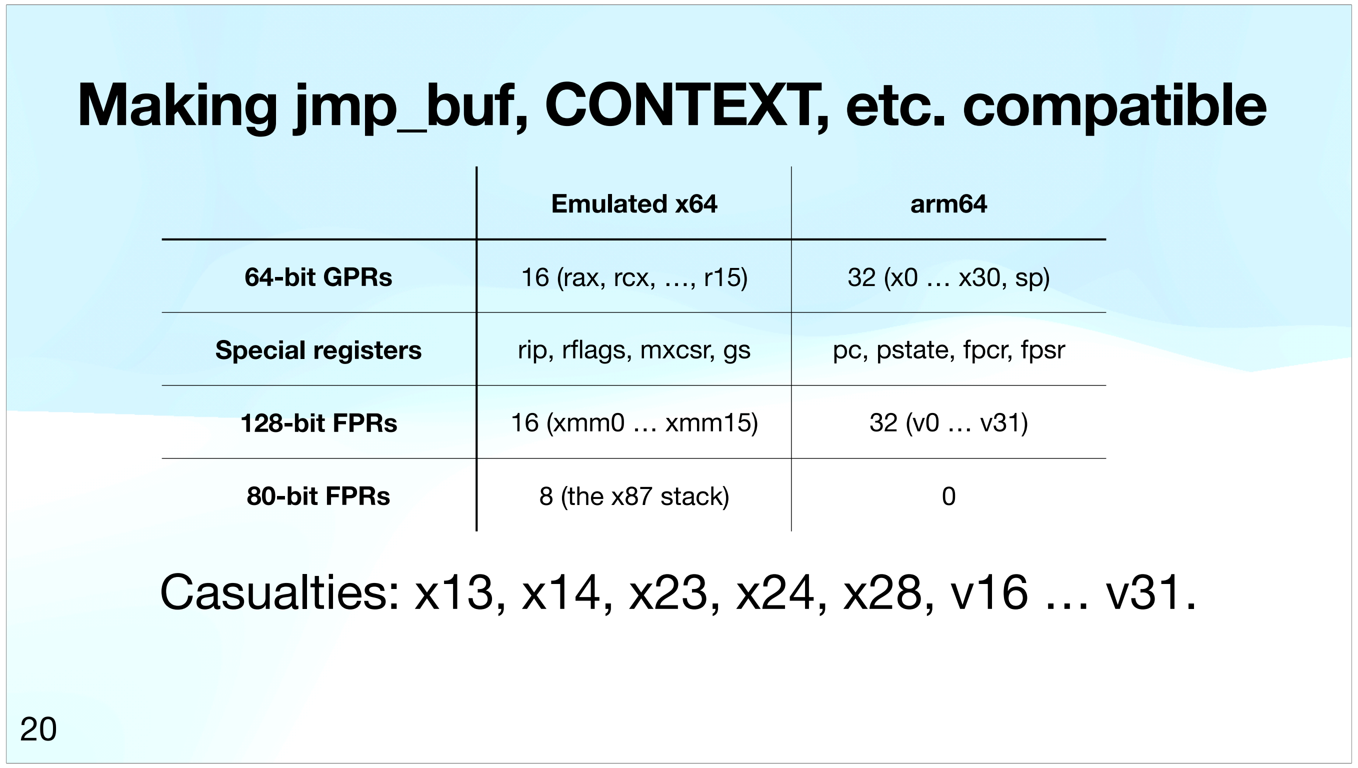
```
struct my_exception_state {
  jmp_buf on_error_jump_to;
};
```

Sharing of data structures actually turns out to be not so simple. C programmers in the room might recall setjmp and longjmp, which are C’s equivalent to throw and catch. The associated jmp_buf type contains the CPU state to be restored when catching. The Arm64EC jmp_buf needs to be the same size/alignment as an x64 jmp_buf, even though there’s a different amount of CPU state involved.

Shared data structure layouts (3)

```
struct my_thread_state {
  CONTEXT ctx;
};
```

There’s also a Windows-specific type called CONTEXT, which contains the entire CPU state for a thread. So again an Arm64EC CONTEXT needs to be the same size/alignment as an x64 CONTEXT, despite having a different amount of CPU state.

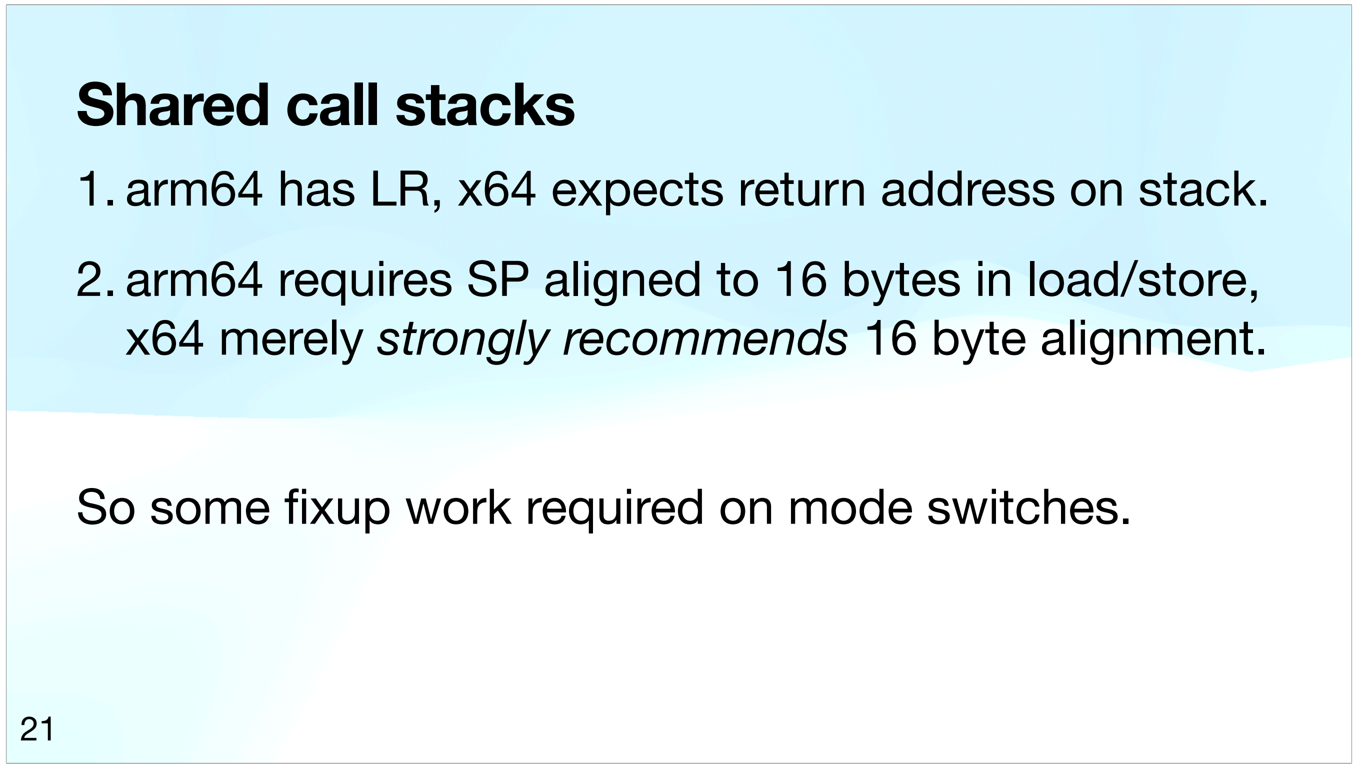


Making jmp_buf, CONTEXT, etc. compatible

	Emulated x64	arm64
64-bit GPRs	16 (rax, rcx, …, r15)	32 (x0 … x30, sp)
Special registers	rip, rflags, mxcsr, gs	pc, pstate, fpcr, fpsr
128-bit FPRs	16 (xmm0 … xmm15)	32 (v0 … v31)
80-bit FPRs	8 (the x87 stack)	0

Casualties: x13, x14, x23, x24, x28, v16 … v31.

We've got the make the column on the right fit into the column on the left. Starting with the first row, x64 has 16 GPRs, whereas arm64 has 32. This is not a good start, as 32 cannot fit into 16. The next row is better; pc can fit into rip, pstate can fit into rflags, fpcr/fpsr can fit into mxcsr. GS is used to store the thread environment block pointer (Linux uses FS for this purpose, Windows uses GS), so we can fit something of our choice into GS (in practice x18, as arm64 uses that as the thread environment block pointer). Moving on to the next row, 32 FPRs cannot fit into 16 FPRs, so we've got a problem here too. Note that the x64 emulator doesn't support AVX or AVX2 or AVX512 (good old patents), hence why there are only 16 FPRs and they're only 128 bits wide. Moving on to the last row, there's finally some good news: x64 has a bunch of 80-bit registers, which have no equivalent in arm64, so we can use those 640 bits to instead store 10 GPRs. When all is said and done, we can fit 27 arm64 GPRs into the x64 state: 16 as GPRs, 10 as 80-bit GPRs, and 1 in GS. That means 5 arm64 GPRs need to be chosen as casualties; there's no space for them in the data structures, so Arm64EC code is disallowed from using them. Microsoft chose to sacrifice x13, x14, x23, x24, and 28. On the FPR side, 16 FPRs need to sacrificed; there's no space for v16 through v31, so again Arm64EC code cannot use them.

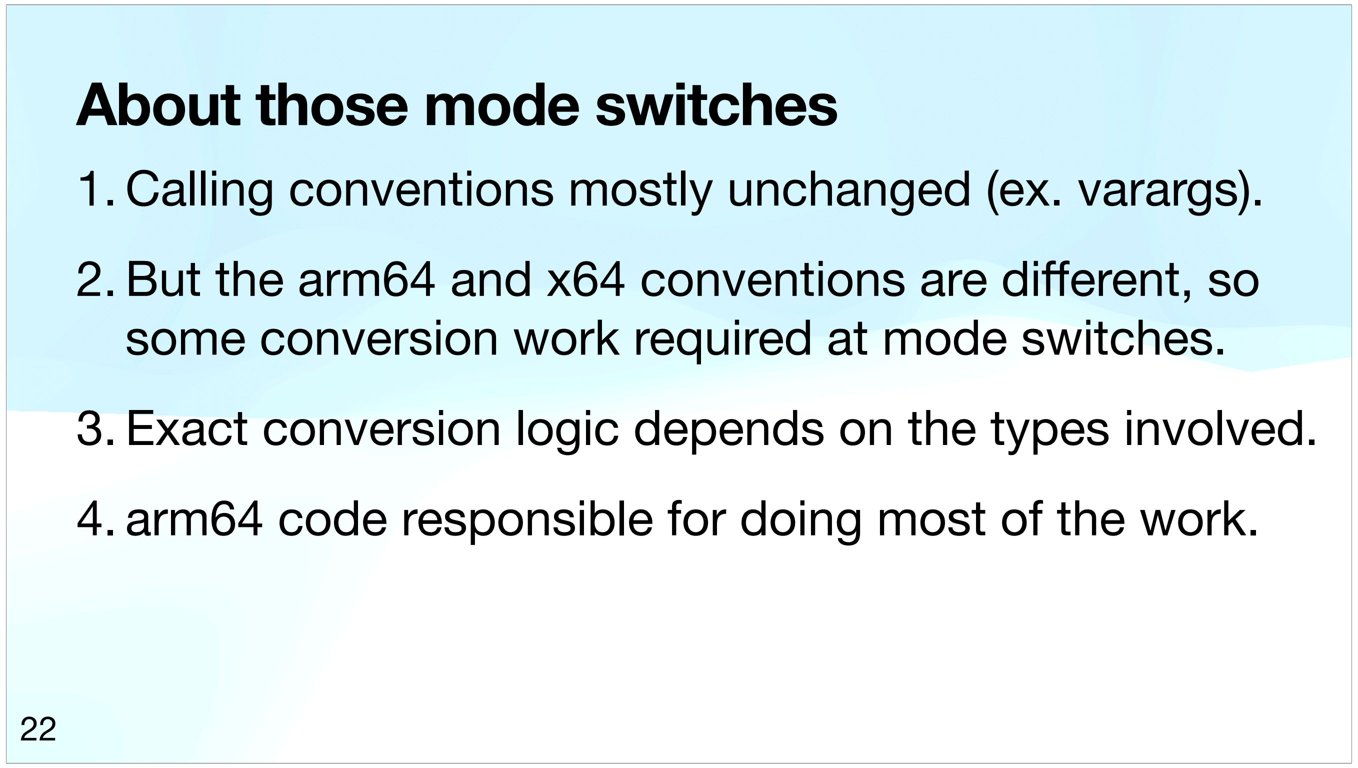


Shared call stacks

1. arm64 has LR, x64 expects return address on stack.
2. arm64 requires SP aligned to 16 bytes in load/store, x64 merely *strongly recommends* 16 byte alignment.

So some fixup work required on mode switches.

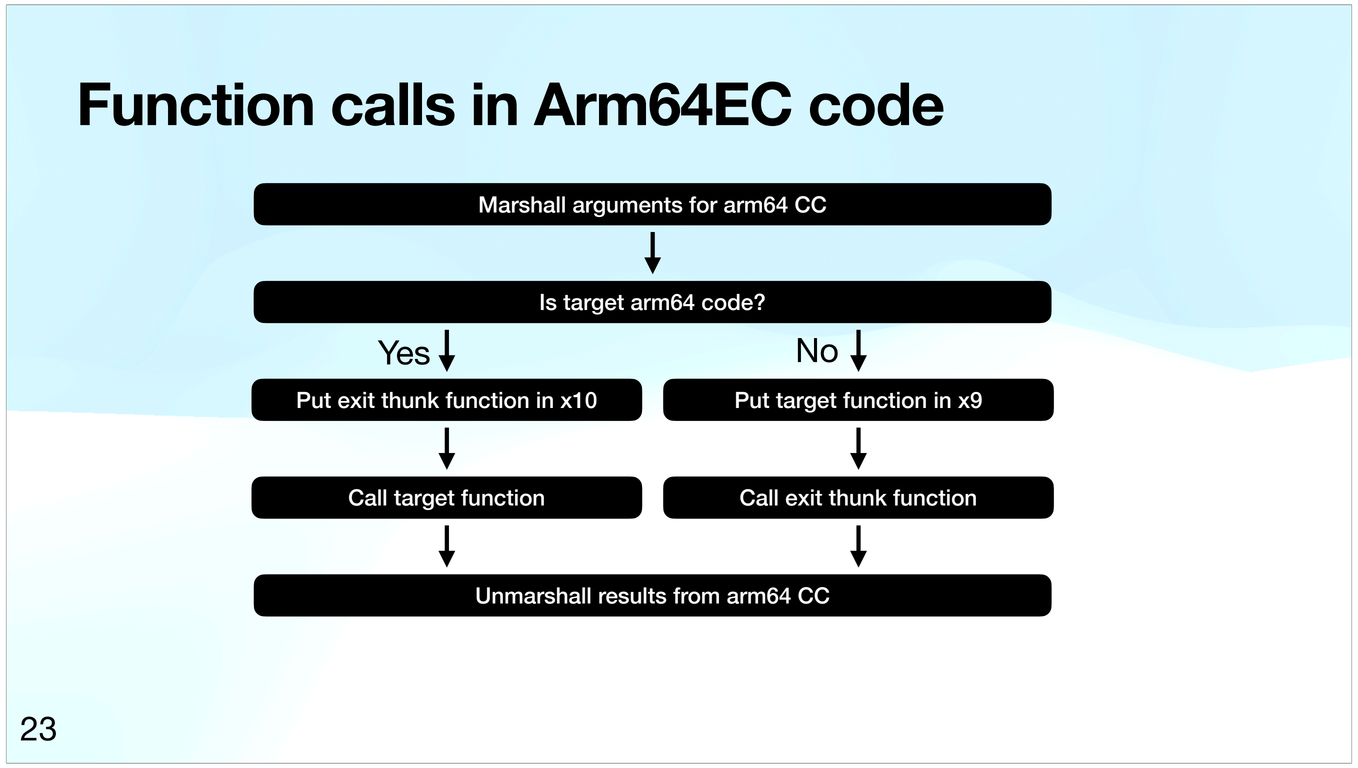
Moving on to shared call stacks, the obvious impedance mismatch is that arm64 uses a register called LR for function call return addresses, whereas x64 puts the return address on the stack. The less obvious mismatch is stack alignment; arm64 requires that SP is 16-byte aligned when used as a base address in loads or stores, and will throw an exception if this isn't the case. Meanwhile the x64 calling convention technically requires 16-byte alignment of stacks, but this isn't enforced, and 8-byte alignment will often work without causing any issues.



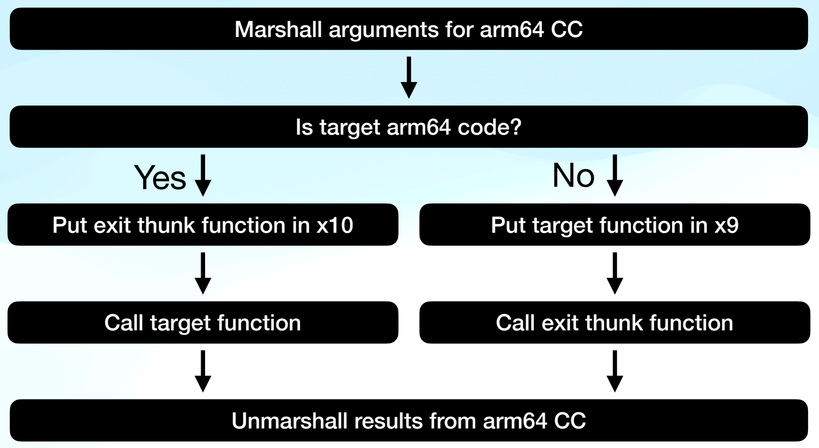
About those mode switches

1. Calling conventions mostly unchanged (ex. varargs).
2. But the arm64 and x64 conventions are different, so some conversion work required at mode switches.
3. Exact conversion logic depends on the types involved.
4. arm64 code responsible for doing most of the work.

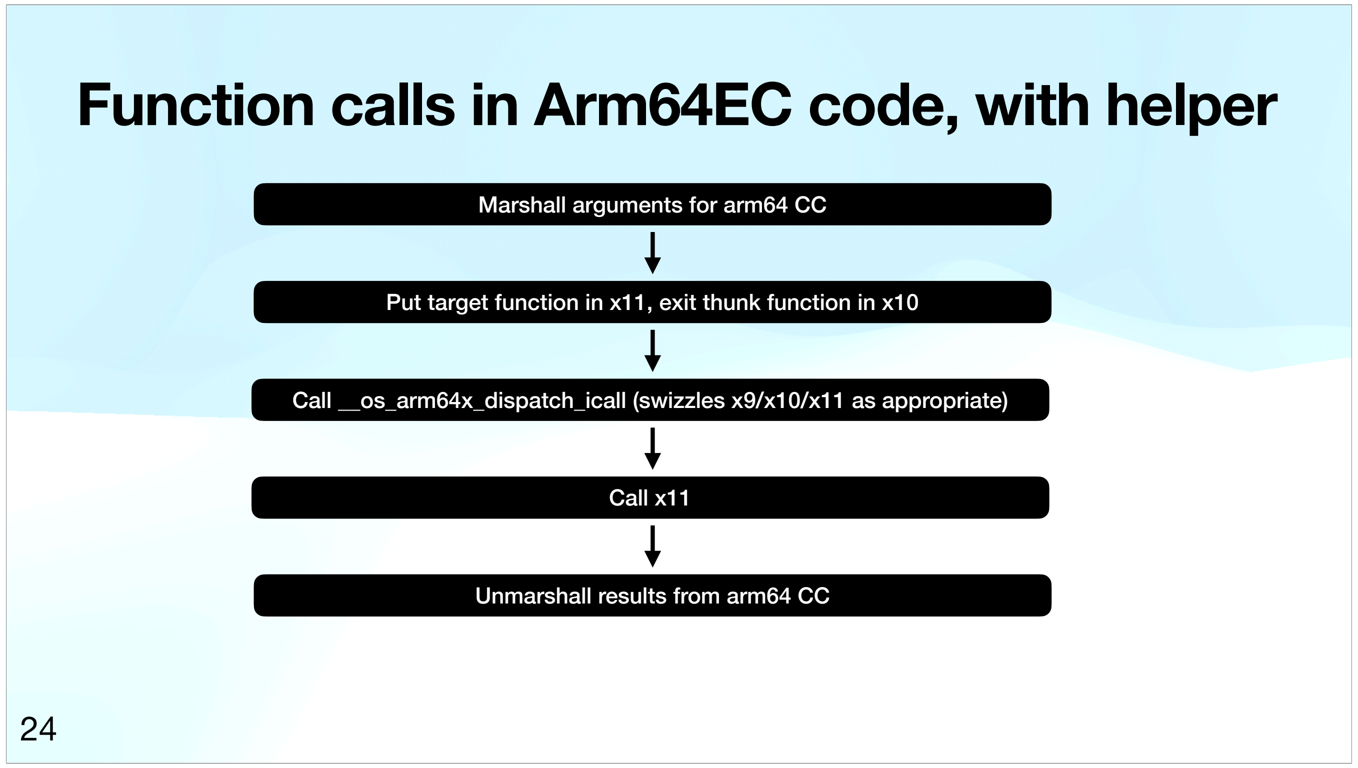
That brings us to mode switches; the actual messy part where control flow switches from arm64 to x64 or vice-versa. The function calling conventions go largely unchanged (except for varargs), but the arm64 and x64 conventions diverge in the details, so a bunch of logic is required to convert from one to the other. The logic depends on the exact function signature (i.e. the return type and the list of argument types). This logic needs to live somewhere, and it can't live in the x64 code, as the whole point is emulating unmodified x64 code, so it has to live in arm64 code.



Function calls in Arm64EC code


Marshall arguments for arm64 CC
Is target arm64 code?
Yes
No
Put exit thunk function in x10
Put target function in x9
Call target function
Call exit thunk function
Unmarshall results from arm64 CC

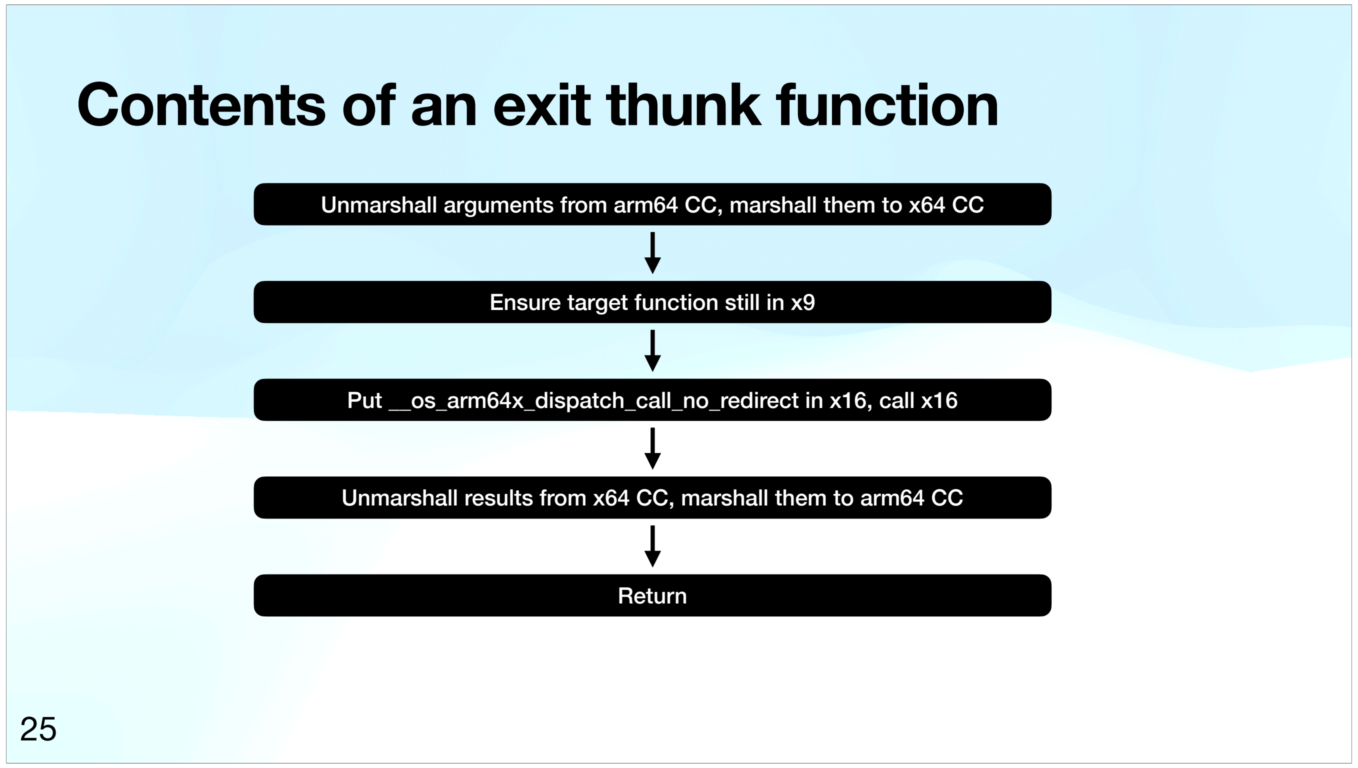
That brings us to how function calls work in Arm64EC code. The left hand column is a regular arm64 function call, albeit with a subtle step about x10 that we'll come to later. The right hand column is the interesting case, and happens when the target is x64 code. It looks similar, except for calling a so-called exit thunk function rather than the actual target function. We'll go into exit thunks shortly, but first, everyone prefers straight-line code to code with branches, and there's a fairly simple way of making this straight-line: lift the initialisation of x9 and x10 above the branch, and then conditionally select which one to call.



Function calls in Arm64EC code, with helper

Marshall arguments for arm64 CC
Put target function in x11, exit thunk function in x10
Call __os_arm64x_dispatch_icall (swizzles x9/x10/x11 as appropriate)
Call x11
Unmarshall results from arm64 CC

This is such a common pattern that Microsoft provide a helper function for it, leading to this straight-line code for doing the same thing. If you ever need to look up what __os_arm64x_dispatch_icall does, it makes the flow chart on this slide equivalent to the flow chart on the previous slide. In practice this slide is what you'll see, but I'll switch back to the previous slide for the sake of a less subtle exposition.



Contents of an exit thunk function

Unmarshall arguments from arm64 CC, marshall them to x64 CC

↓

Ensure target function still in x9

↓

Put __os_arm64x_dispatch_call_no_redirect in x16, call x16

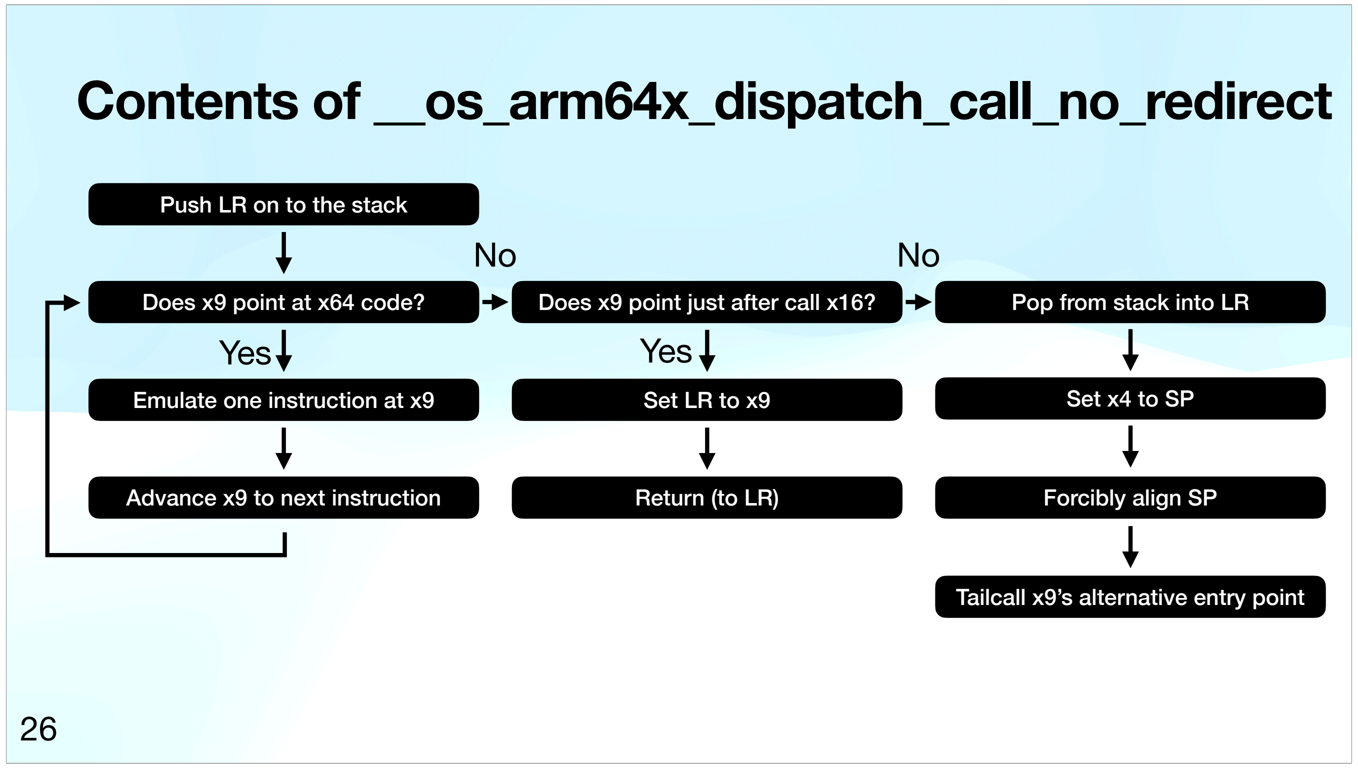
↓

Unmarshall results from x64 CC, marshall them to arm64 CC

↓

Return

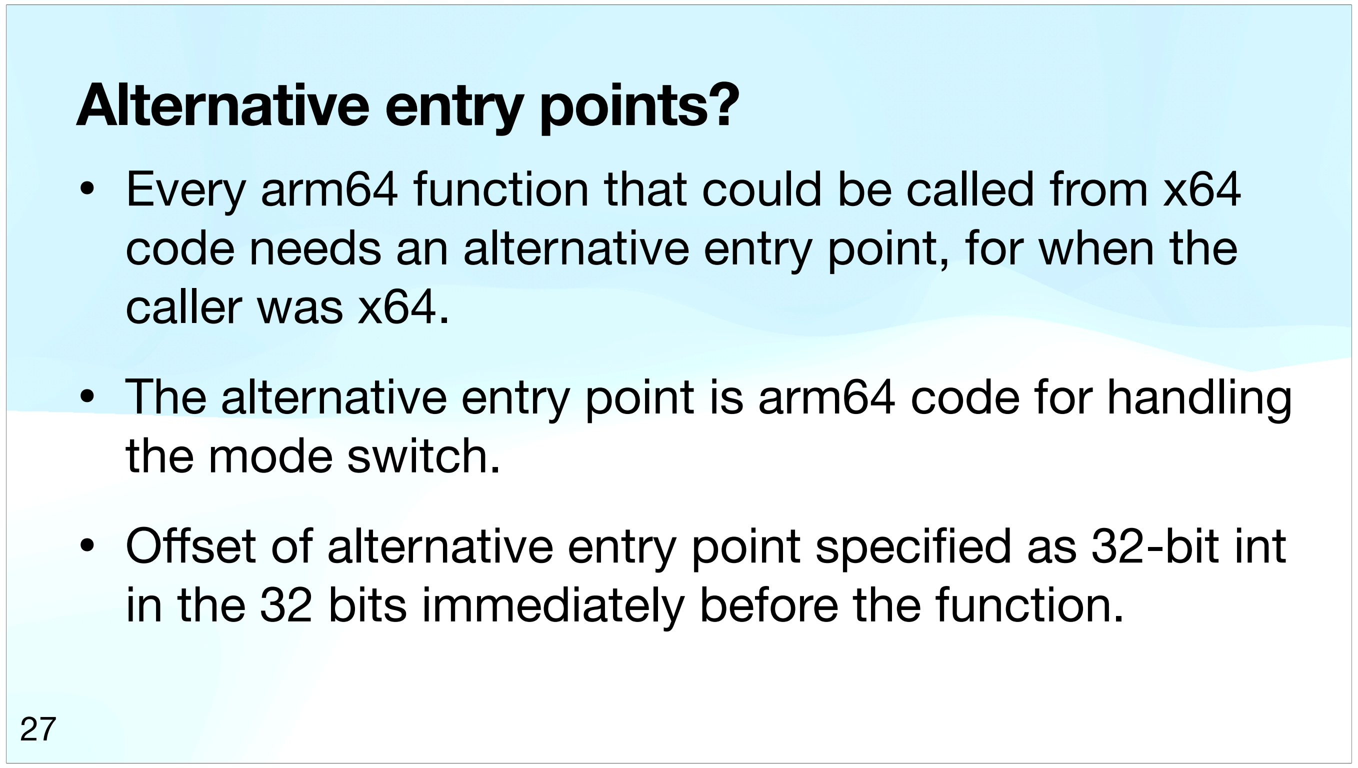
What does an exit thunk do? It is the missing part of the diagram required to transfer control to the x64 emulator: convert the arguments from one calling convention to another, then call the emulator, then convert the results from one calling convention to another, then return. The only subtle part is that the call to the emulator has to done as calling the x16 register. We’ll see why in a bit.



Contents of __os_arm64x_dispatch_call_no_redirect

Push LR on to the stack
Does x9 point at x64 code?
Yes
Emulate one instruction at x9
Advance x9 to next instruction
No
Does x9 point just after call x16?
Yes
Set LR to x9
Return (to LR)
No
Pop from stack into LR
Set x4 to SP
Forcibly align SP
Tailcall x9's alternative entry point

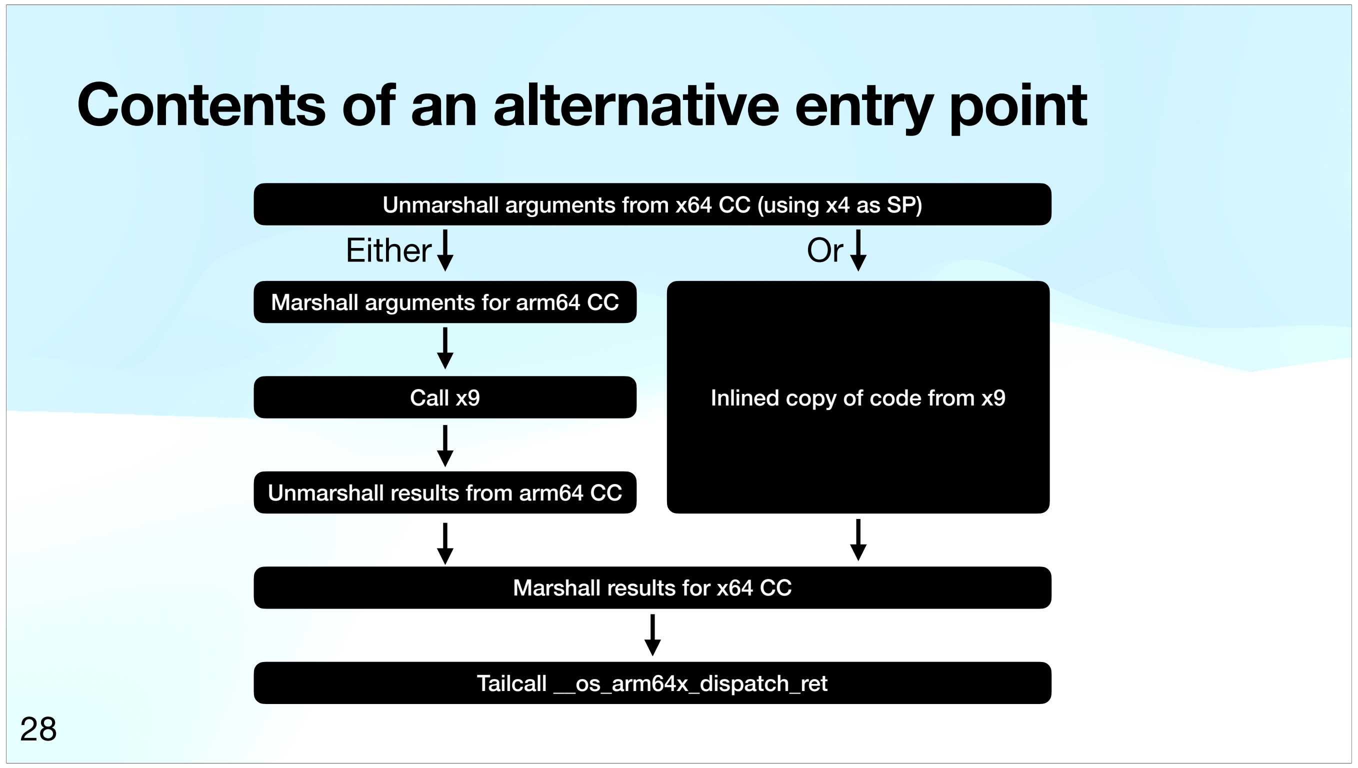
The next layer of the onion is what the emulator does. The left hand column is a basic emulator loop, emulating one x64 instruction at a time. In practice the loop is more complex, translating more than one instruction at a time, sending translation requests to an out-of-process caching AOT compiler, and a bunch of other tricks, but for exposition we can pretend it is a simple loop. At some point the instruction pointer stops pointing at x64 code, which implies that the emulated code has either just returned to arm64 code, or just called arm64 code. To determine which, the emulator looks at the four bytes prior the new instruction pointer. If those bytes contain a call x16 instruction, then we're doing a return. This is why the prior slide required calls into the emulator to be call x16. The middle column is the logic for doing a return to arm64, which is super simple, because the exit thunk does most of the work. The final column on the right is for x64 code calling arm64 code, which is some stack fixup, followed by a tailcall to arm64 code.



Alternative entry points?

- Every arm64 function that could be called from x64 code needs an alternative entry point, for when the caller was x64.
- The alternative entry point is arm64 code for handling the mode switch.
- Offset of alternative entry point specified as 32-bit int in the 32 bits immediately before the function.

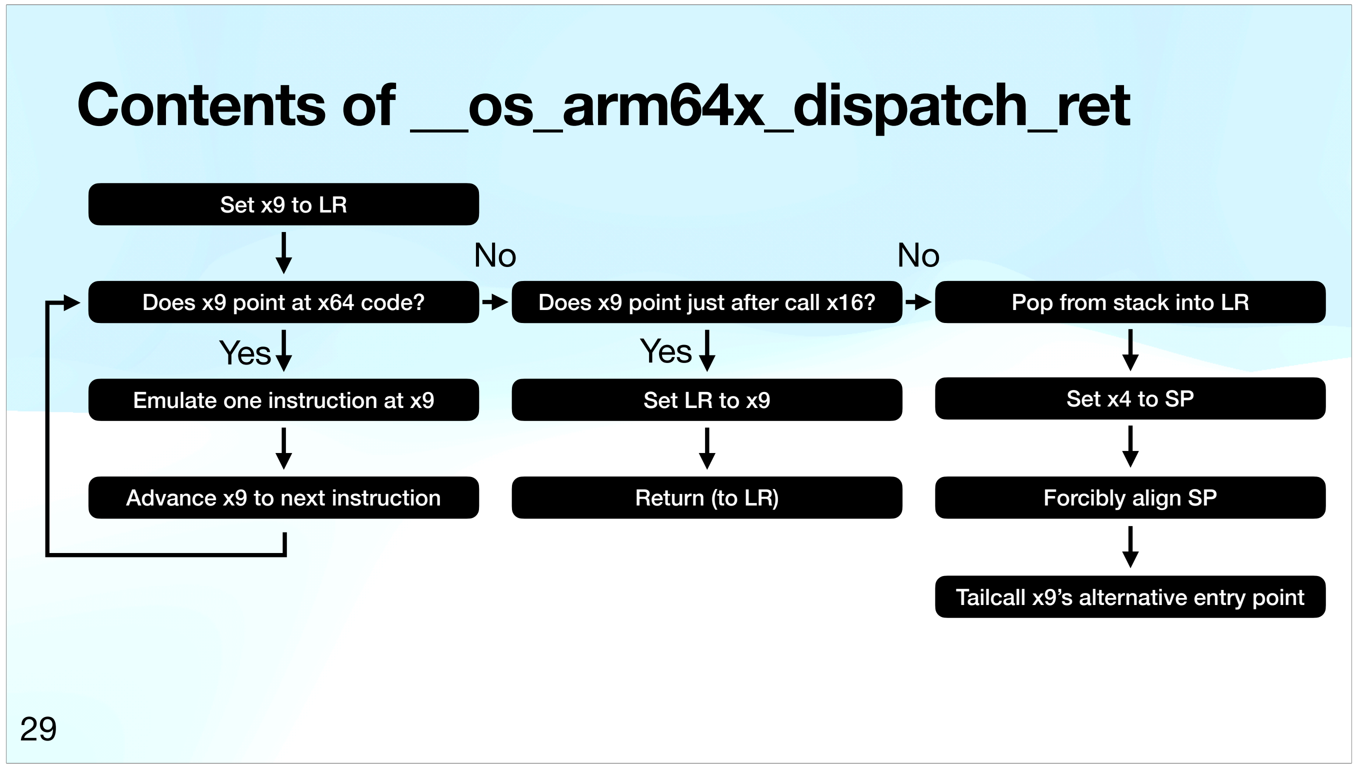
I pulled a fast one with the previous slide, and so you’re probably asking what an alternative entry point is.



Contents of an alternative entry point

Unmarshall arguments from x64 CC (using x4 as SP)
Either
Or
Marshall arguments for arm64 CC
Call x9
Unmarshall results from arm64 CC
Inlined copy of code from x9
Marshall results for x64 CC
Tailcall __os_arm64x_dispatch_ret

In practice, alternative entry points are like exit thunks, but in reverse: convert the arguments from the x64 calling convention to the arm64 calling convention (using x4 as stack pointer, because of the forced re-alignment on the previous slide), then call the primary entry point, then convert the results back to the x64 calling convention. The logic here depends on the type signature of the function, and so several distinct functions can all share the same alternative entry point, provided that they have the same type. Alternatively, if the alternative entry point is only used by a single function, and the code for that function is small, it could be inlined, as in the right hand side.



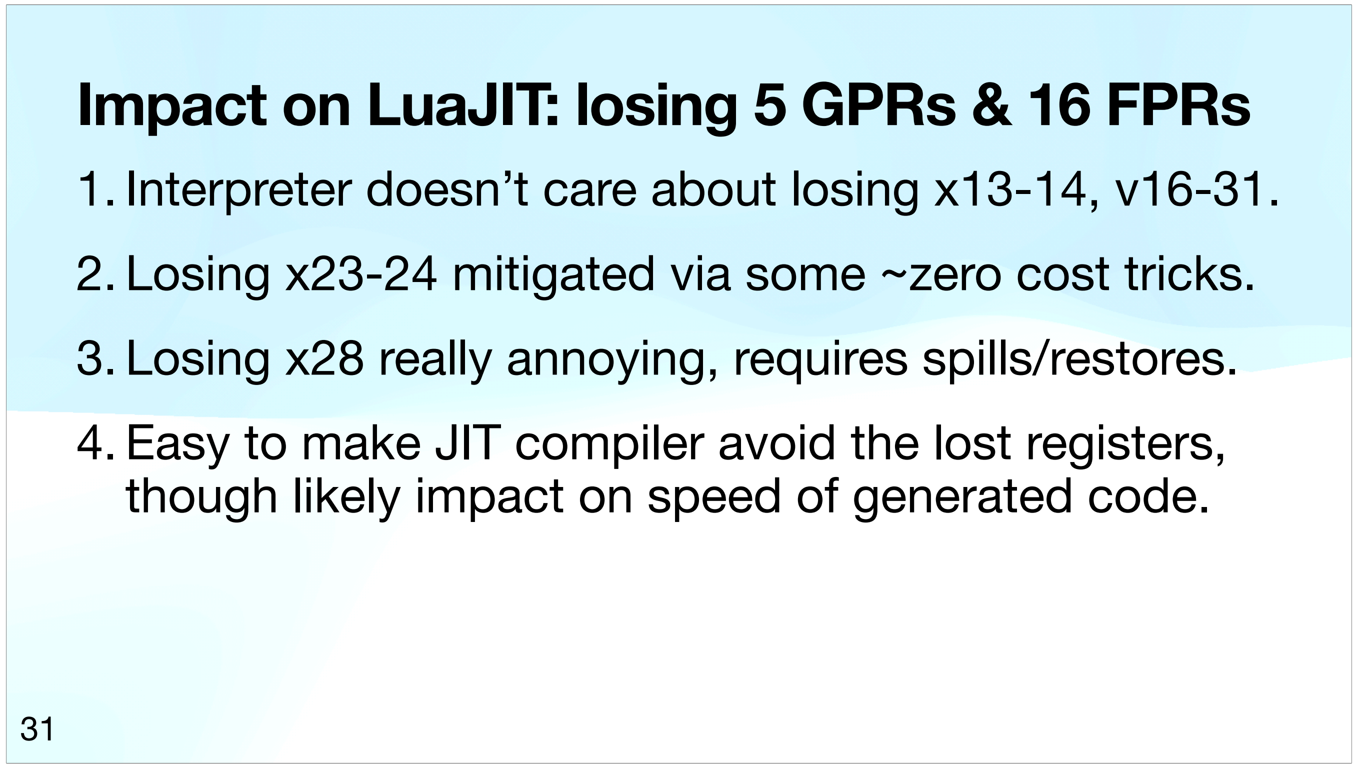
Contents of __os_arm64x_dispatch_ret

Set x9 to LR
Does x9 point at x64 code?
No
Does x9 point just after call x16?
No
Pop from stack into LR
Yes
Emulate one instruction at x9
Advance x9 to next instruction
Yes
Set LR to x9
Return (to LR)
Set x4 to SP
Forcibly align SP
Tailcall x9's alternative entry point

That leads us to __os_arm64x_dispatch_ret, which looks complex, but don't worry, this is exactly the same as slide 26, except for the first box in the top left. For this first box, we can trace back what LR is, and it'll come from "Pop from stack into LR" on slide 26, which is what runs after x64 does a function call into arm64, and x64 calls involve pushing the return address onto the stack, meaning LR at this point will be that x64 return address.

Agenda

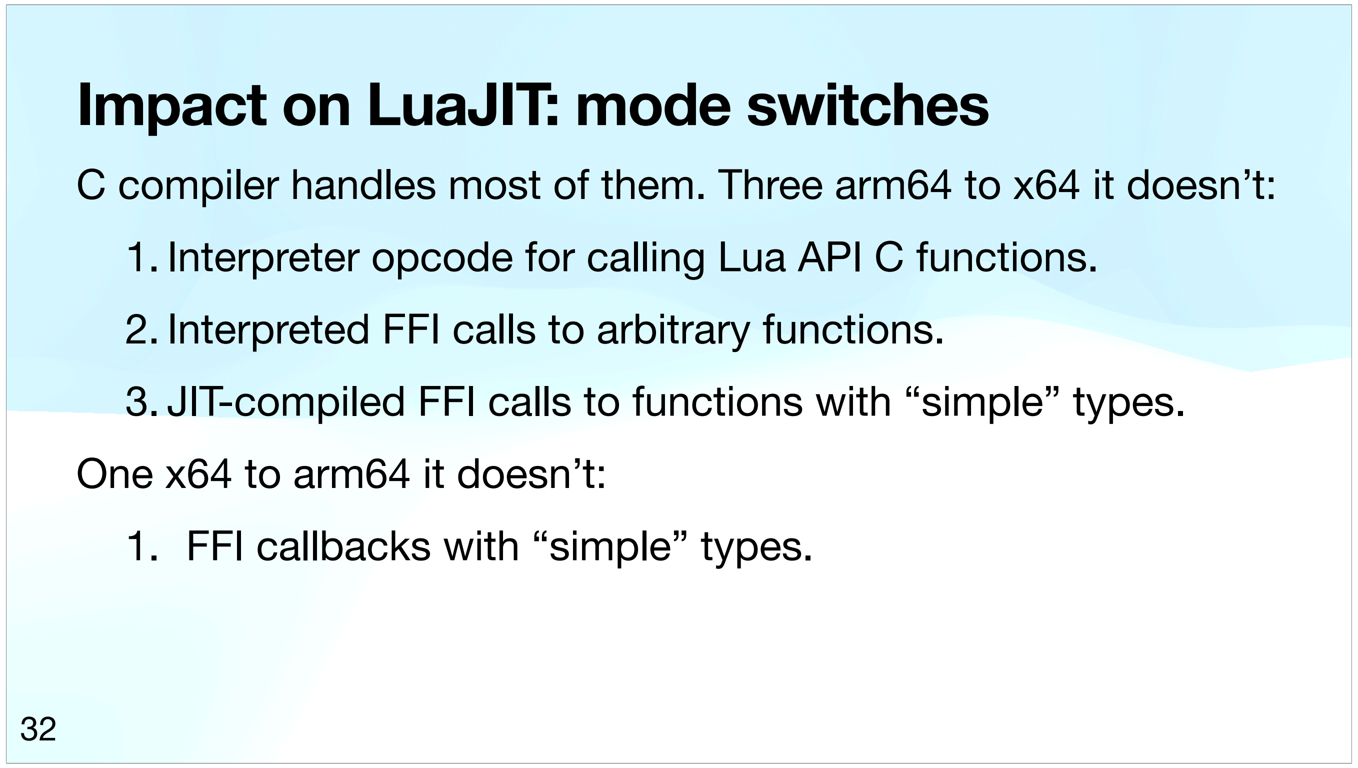
1. General landscape of emulating x64 on arm64.
2. What is Arm64EC?
3. Lessons learnt porting LuaJIT to Arm64EC. ←



Impact on LuaJIT: losing 5 GPRs & 16 FPRs

1. Interpreter doesn't care about losing x13-14, v16-31.
2. Losing x23-24 mitigated via some ~zero cost tricks.
3. Losing x28 really annoying, requires spills/restores.
4. Easy to make JIT compiler avoid the lost registers, though likely impact on speed of generated code.

LuaJIT is written in a mixture of assembly and C, notably the interpreter is several thousand lines of assembly. Porting that assembly code to Arm64EC means it can no longer use certain registers. In this particular area, porting the JIT compiler was easier than porting the interpreter, as the JIT already had support for avoiding certain registers.



Impact on LuaJIT: mode switches

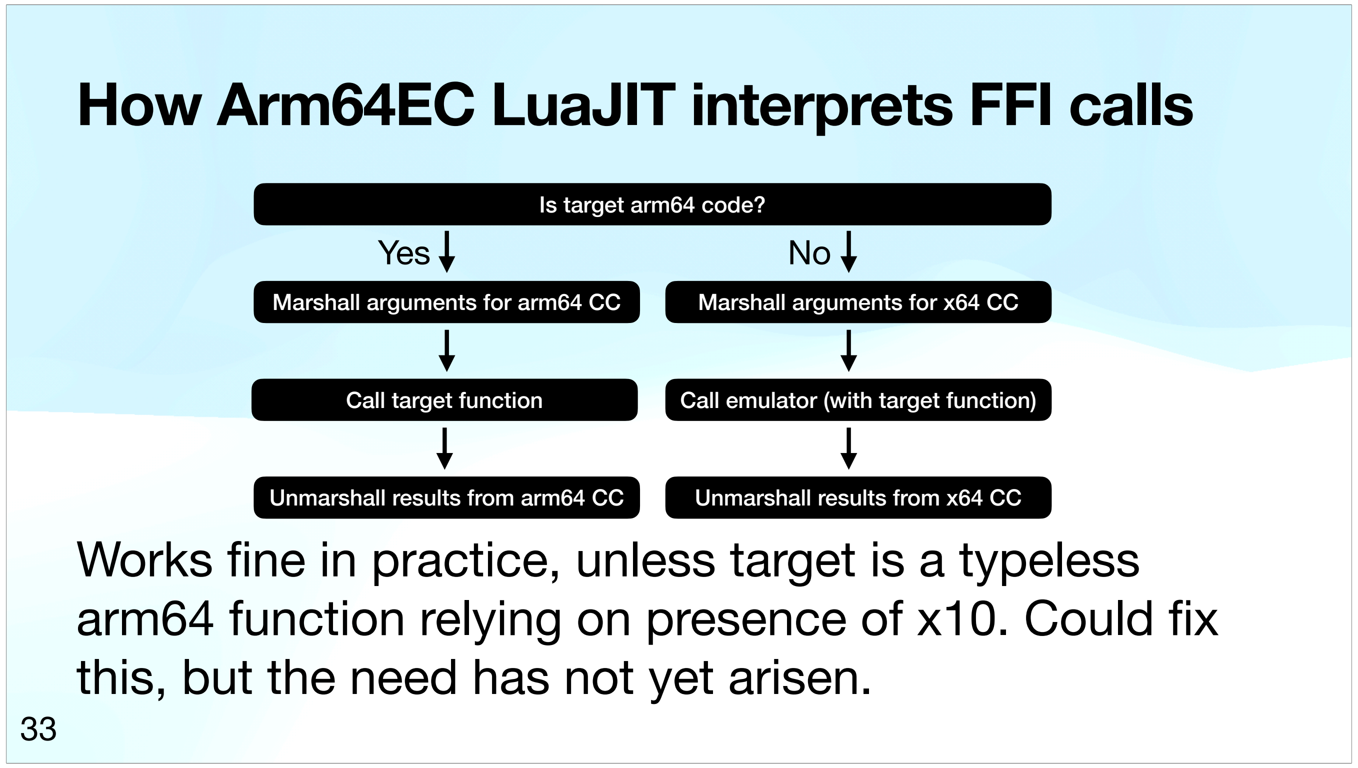
C compiler handles most of them. Three arm64 to x64 it doesn’t:

1. Interpreter opcode for calling Lua API C functions.
2. Interpreted FFI calls to arbitrary functions.
3. JIT-compiled FFI calls to functions with “simple” types.

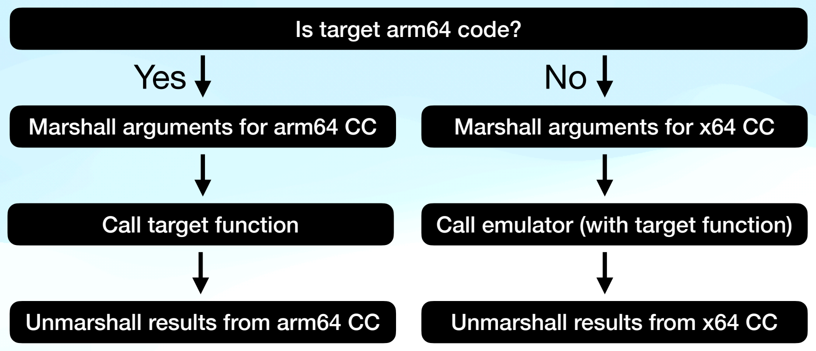
One x64 to arm64 it doesn’t:

1. FFI callbacks with “simple” types.

Handling mode switches was the other major hard part of porting. The C compiler handles most of the work, but there’s assembly code for the interpreter and for the FFI that needed porting manually. Thankfully there’s only one place in the interpreter that can call out to a completely arbitrary function, and the type signature of that call is as simple as it can get. The FFI turns out to be more annoying though.

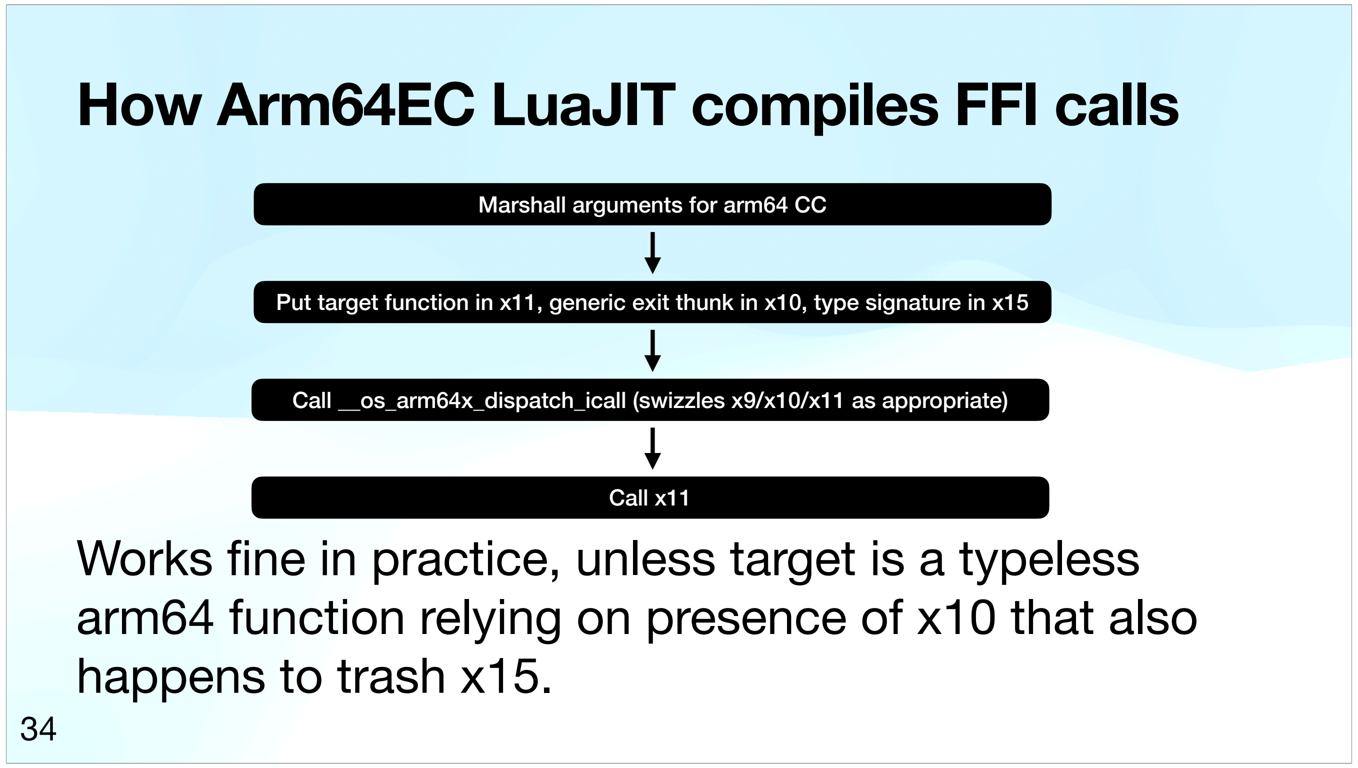


How Arm64EC LuaJIT interprets FFI calls


Is target arm64 code?
Yes
No
Marshall arguments for arm64 CC
Marshall arguments for x64 CC
Call target function
Call emulator (with target function)
Unmarshall results from arm64 CC
Unmarshall results from x64 CC

Works fine in practice, unless target is a typeless arm64 function relying on presence of x10. Could fix this, but the need has not yet arisen.

Interpreted FFI calls are one-shot calls to a particular address, so we can look at that address, determine what architecture it is, marshall the arguments as appropriate for it, then call it, then unmarshall the results as appropriate. This looks like a nice efficient fusion of a regular call with an exit thunk, and works fine in practice, but doesn't quite follow the rules. Why not? If going down the left hand column, we call the target function without providing an exit thunk pointer in x10. Following the rules would be hard to do, as there are an infinite number of type signatures that the FFI could call, so we can't prepare an infinite number of exit thunks ahead of time. Instead we'd need to JIT compile the exit thunks, which is a world of pain I'd rather avoid.

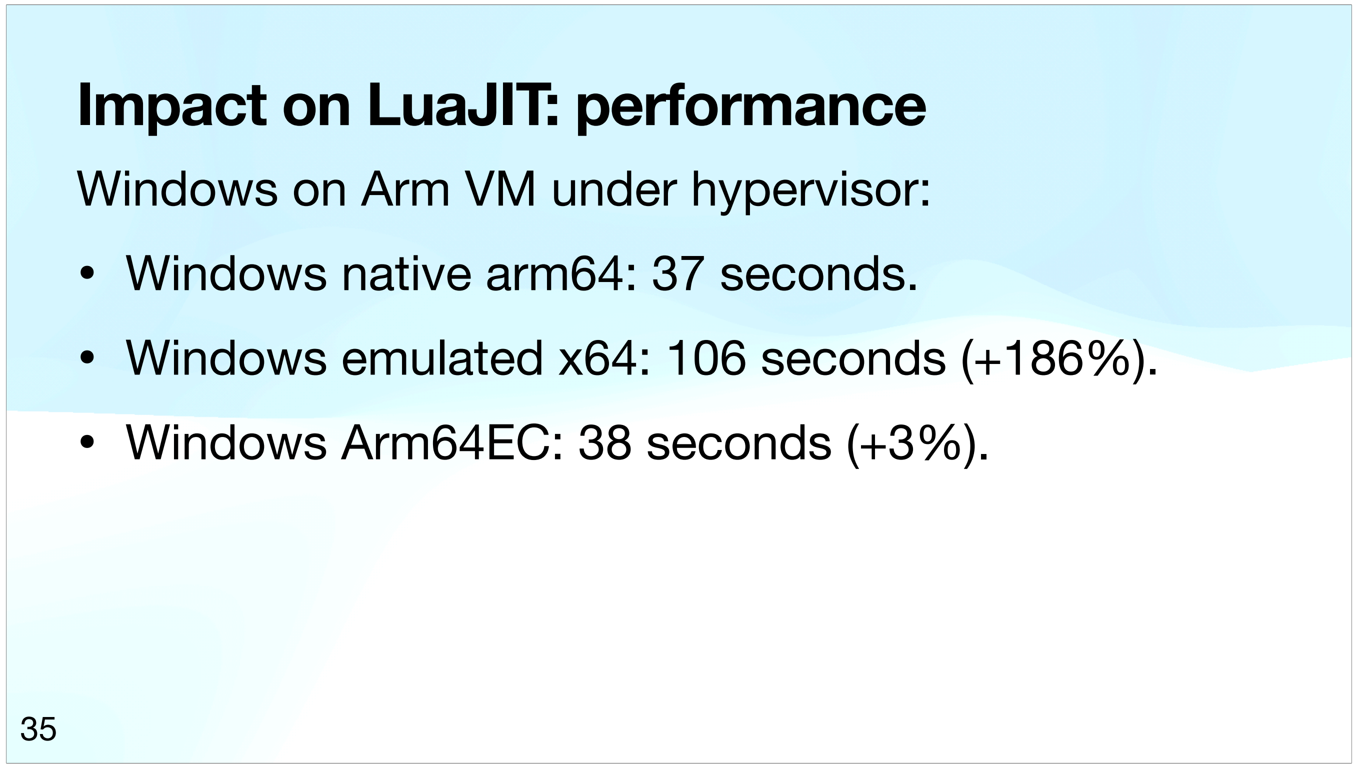


How Arm64EC LuaJIT compiles FFI calls

Marshall arguments for arm64 CC
Put target function in x11, generic exit thunk in x10, type signature in x15
Call __os_arm64x_dispatch_icall (swizzles x9/x10/x11 as appropriate)
Call x11

Works fine in practice, unless target is a typeless arm64 function relying on presence of x10 that also happens to trash x15.

JIT-compiled FFI calls are a different kettle of fish. Whereas interpreted calls are one-shot, compiled calls can be compiled once and then run multiple times. If compiling a call of a function pointer, the value of that function pointer might sometimes be arm64 code and other times be x64 code. This means that we have to be closer to following the rules. Not all type signatures are allowed for JIT-compiled FFI calls, but there's still a huge number of allowed signatures, too many to pre-compile all possible exit thunks. To avoid having to JIT-compile exit thunks, I cheated and wrote a single generic exit thunk that handles all type signatures allowed for JIT-compiled FFI calls, and communicated the type signature to it in a hopefully-unused register.

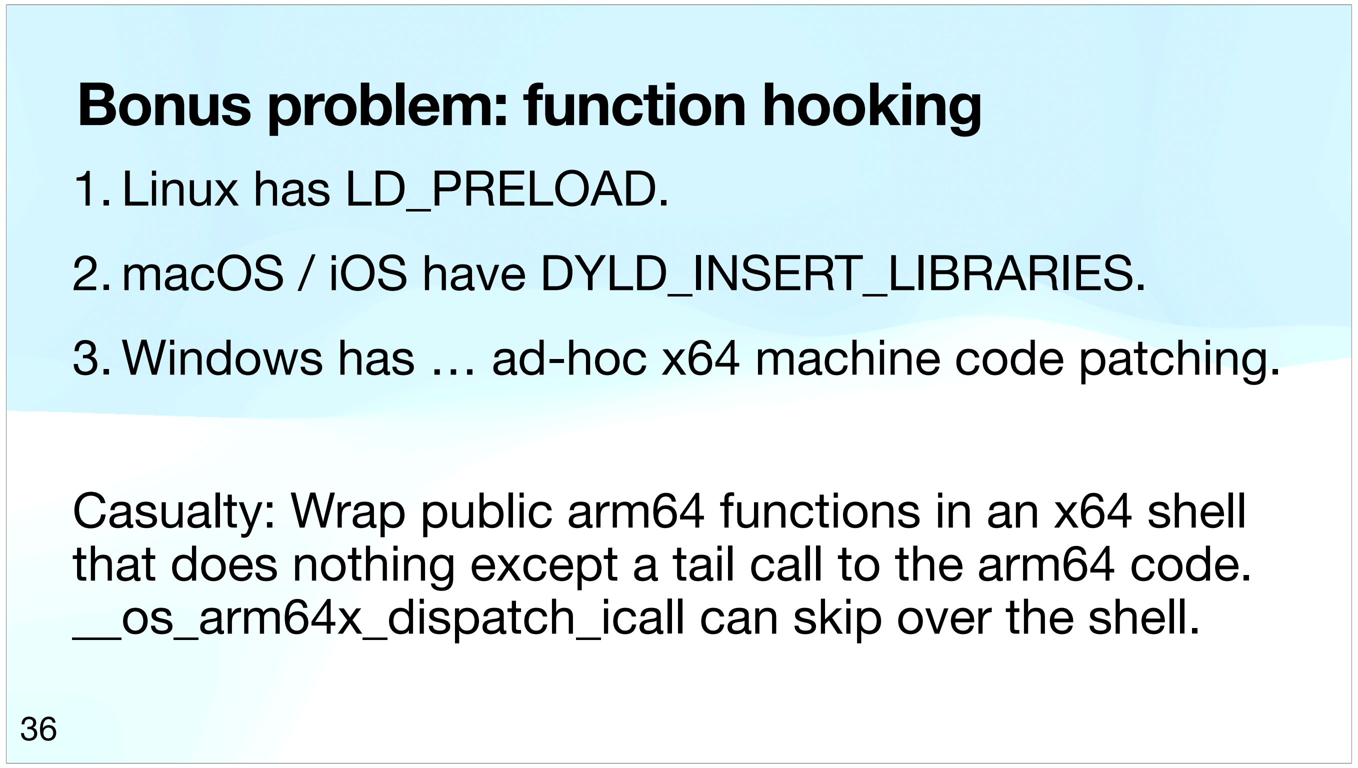


Impact on LuaJIT: performance

Windows on Arm VM under hypervisor:

- Windows native arm64: 37 seconds.
- Windows emulated x64: 106 seconds (+186%).
- Windows Arm64EC: 38 seconds (+3%).

Finally, the slide you've all been waiting for: the motivation for Arm64EC is performance, so does it deliver? I'd say the answer is a resounding yes; the emulated x64 parts are still going to be horribly slow, but the parts ported to arm64 run at basically native speed. In other words, the cost of making arm64 emulation compatible is pretty low.



Bonus problem: function hooking

1. Linux has LD_PRELOAD.
2. macOS / iOS have DYLD_INSERT_LIBRARIES.
3. Windows has … ad-hoc x64 machine code patching.

Casualty: Wrap public arm64 functions in an x64 shell that does nothing except a tail call to the arm64 code. __os_arm64x_dispatch_icall can skip over the shell.

Bonus slide if there's time. A problem you didn't know you had and/or a massive concession to backwards compatibility. Microsoft Research used to sell a product called Detours for doing ad-hoc machine code patching, though make it open source and free in 2016.