VECTORS MEET VIRTUALIZATION

ALEX BENNÉE

FOSDEM 2018

INTRODUCTION

- Alex Bennée
  - alex.bennee@linaro.org
  - stsquad on #qemu
- Virtualization Developer @ Linaro
- Projects:
  - QEMU, KVM, ARM

WHAT IS QEMU?

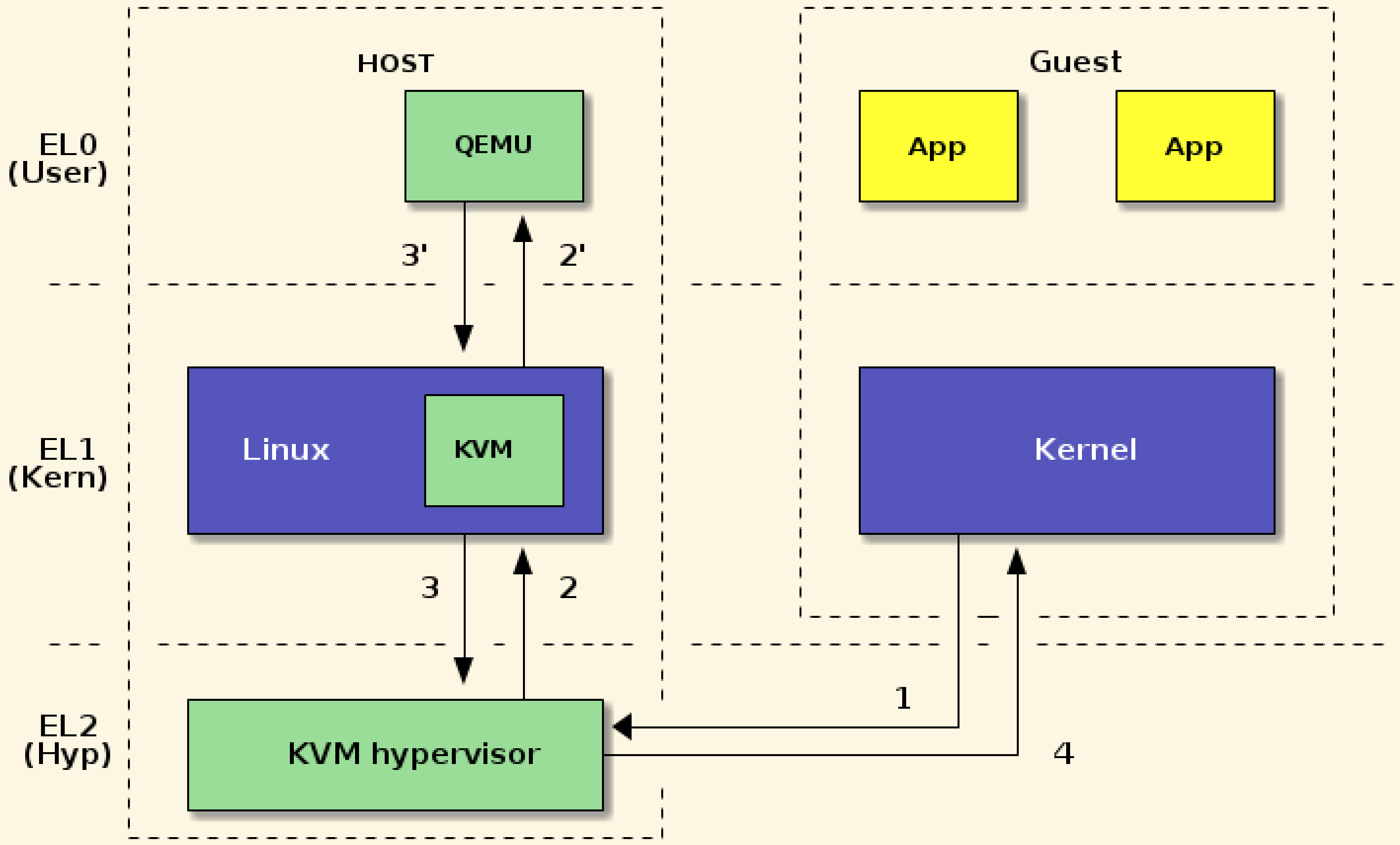
From: www.qemu.org
"QEMU is a generic and open source machine emulator and virtualizer."

TWO TYPES OF VIRTUALIZATION

- Hardware Assisted Virtualization (KVM*)
- Cross Architecture Emulation (TCG)

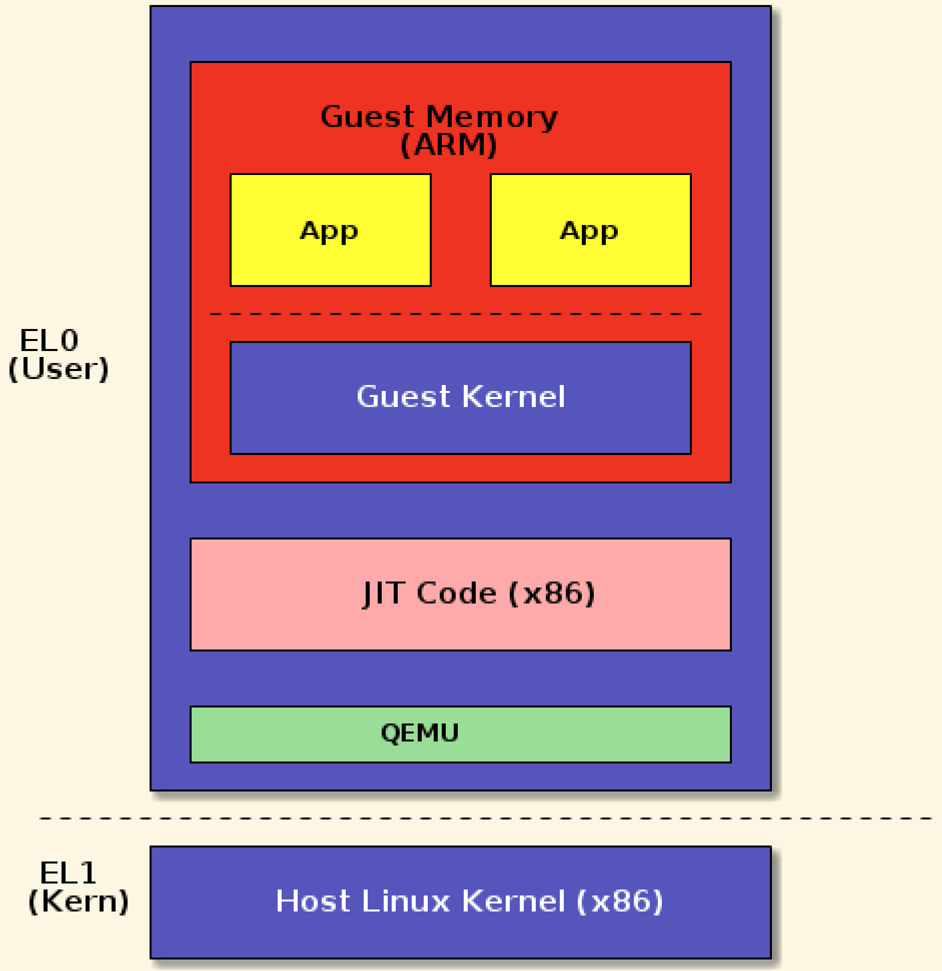
HARDWARE ASSISTED VIRTUALIZATION

High Performance, Cloud, Server Consolidation



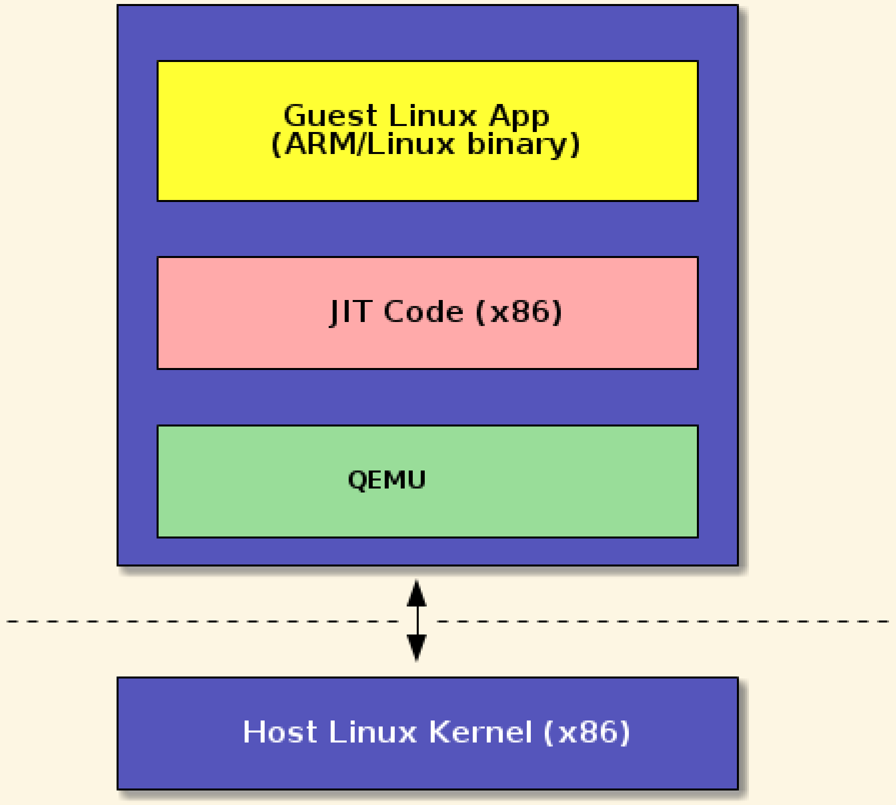
FULL SYSTEM EMULATION

Android Emulator, Embedded Development, New Architectures

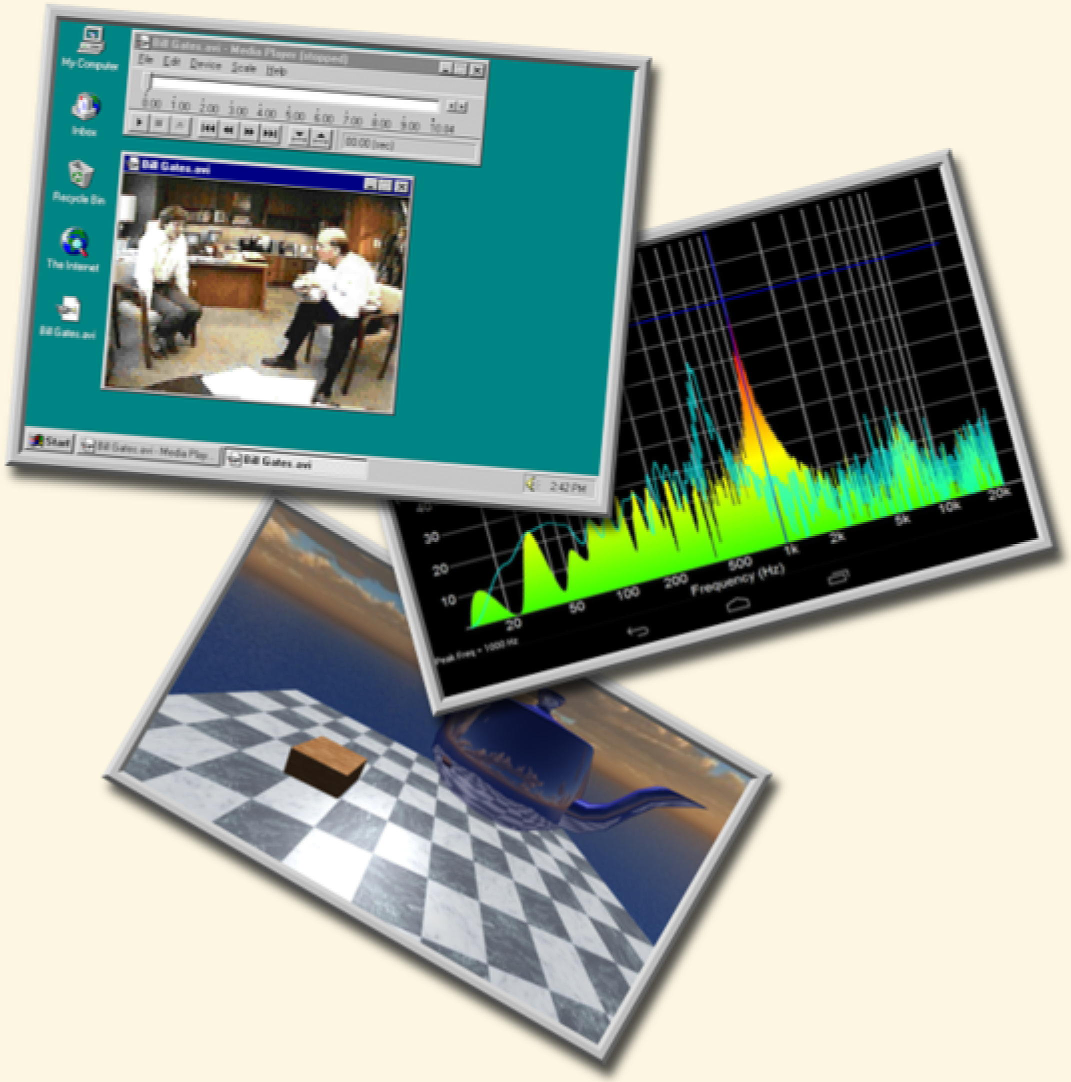


LINUX USER EMULATION

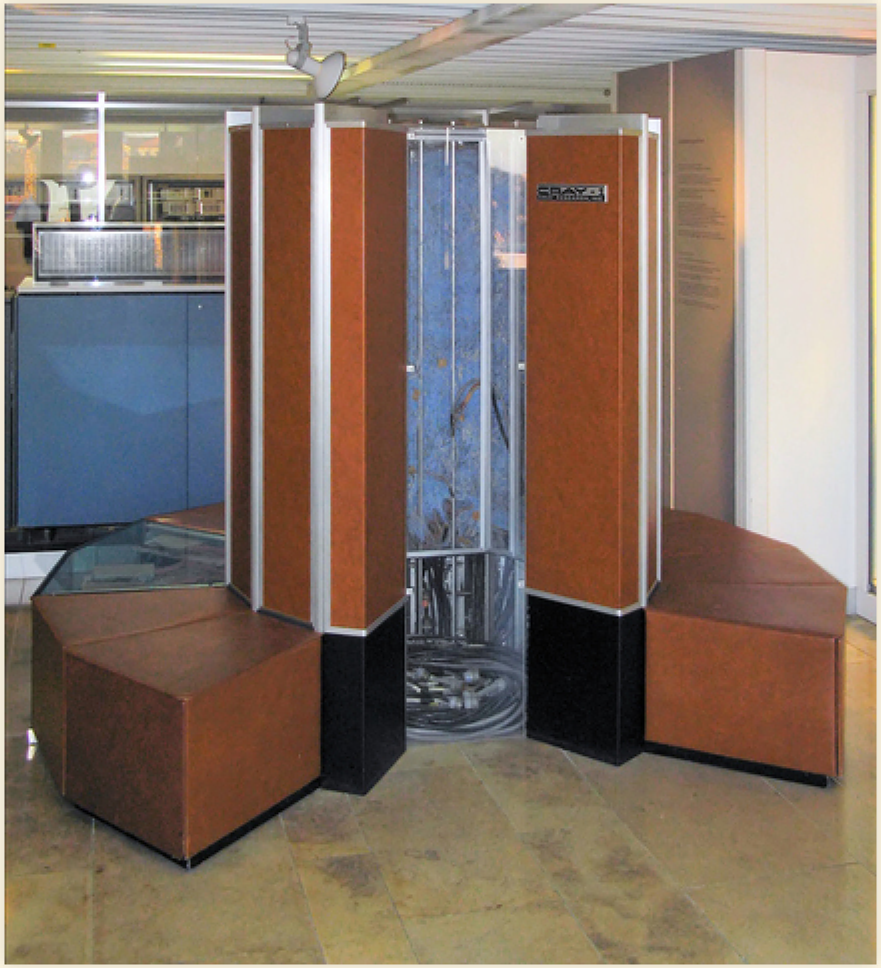
Cross-development tools, Legacy binaries



WHAT ARE VECTORS?


Bill Gates.avi - Media Player (stopped)
File Edit Device Scale Help
Bill Gates.avi
My Computer
Inbox
Recycle Bin
The Internet
Start
2:42 PM
Frequency (Hz)
20 50 100 200 500 1k 2k 5k 10k 20k
10 20 30

HISTORY QUIZ



CRAY 1 SPECS

Addressing	8 24 bit address
Scalar Registers	8 64 bit data
Vector Registers	8 (64x64bit elements)
Clock Speed	80 Mhz
Performance	up to 250 MFLOPS*
Power	250 kW

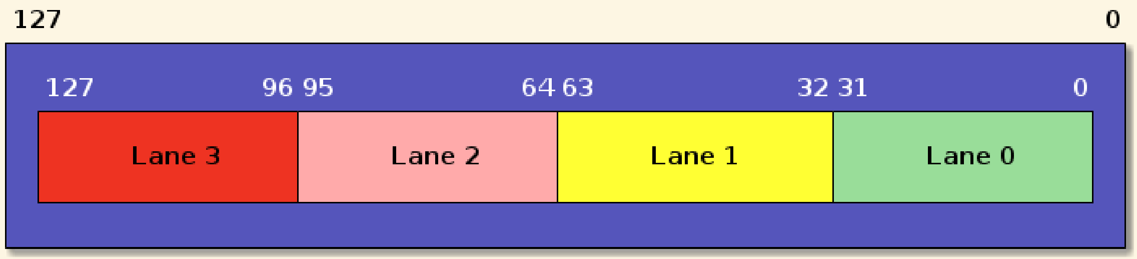
ref: The Cray-1 Computer System, Richard M Russell, Cray Reasearch Inc, ACM Jan 1978, Vol 21, Number 1

ARCHITECTURES WITH VECTORS

Year	ISA
1994	SPARC VIS
1997	Intel x86 MMX
1996	MIPS MDMX
1998	AMD x86 **3DNow!**
2002	PowerPC Altivec
2009	ARM NEON/AdvSIMD

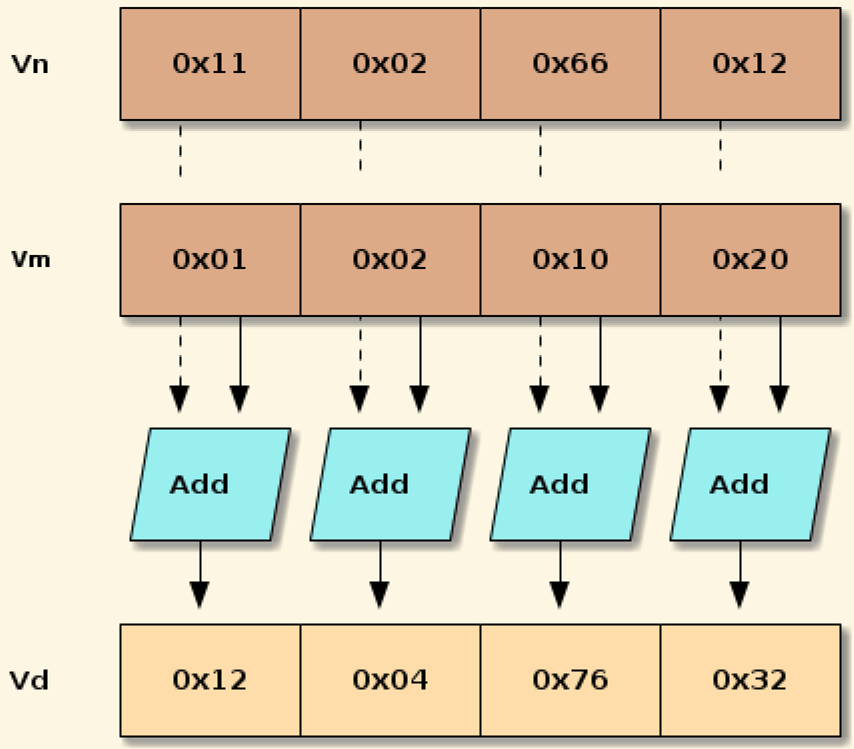
VECTOR REGISTER

128 bit wide, 4 x 32 bit elements



VECTOR OPERATION

```
vadd %Vd, %Vn, %Vm
```



VECTOR SIZE IS GROWING

Year	SIMD ISA	Vector Width	Addressing
1997	MMX	64 bit	2x32/4x16/8x8
2001	SSE2	128 bit	2x64/4x32/8x16/16x8
2011	AVX	256 bit	4x64/8x32
2017	AVX-512	512 bit	8x64/16x32/32x16/64x8

ARM SCALABLE VECTOR EXTENSIONS (SVE)

- IMPDEF vector size (128-2048* bit)
- nx64/2nx32/4nx16/8nx8
- New instructions for size agnostic code

STRCPY (C CODE)

```c
void strcpy(char *restrict dst, const char *src)
{
  while (1) {
    *dst = *src;
    if (*src == '\0') break;
    src++; dst++;
  }
}
```

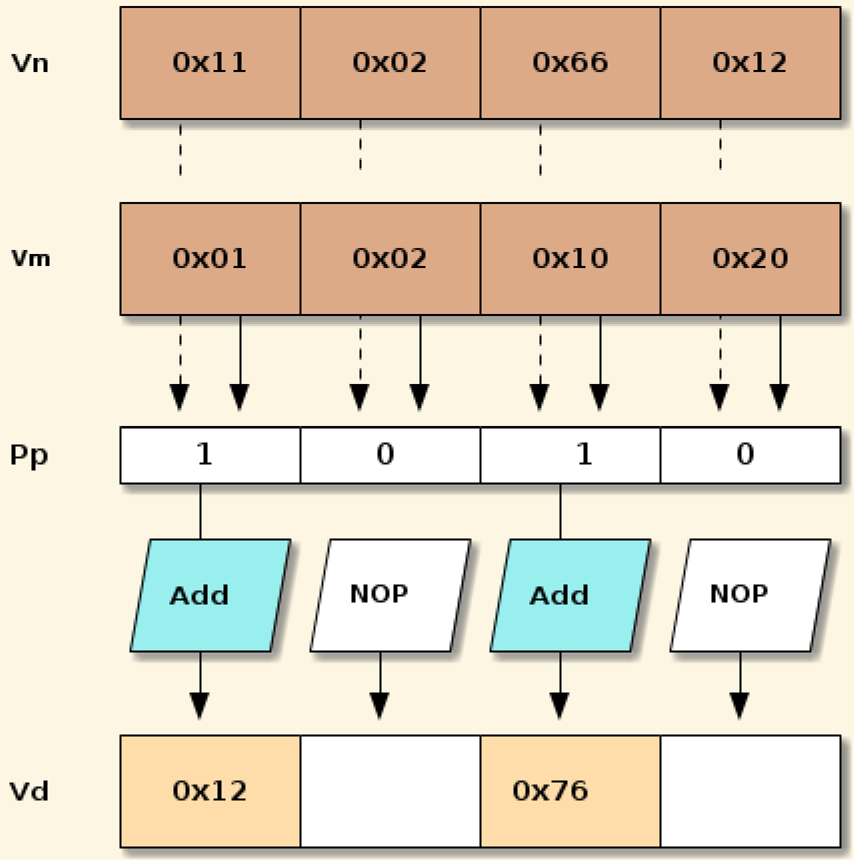
From:
https://developer.arm.com/-/media/developer/developers/hpc/white-papers/a-sneak-peek-into-sve-and-vla-programming.pdf

STRCPY (SVE ASSEMBLY)

```
sve_strcpy:                                     # header
        mov x2, 0
        ptrue p2.b
loop:                                           # loop body
        setffr                                  # set first fault register
        ldff1b z0.b, p2/z, [x1, x2]
        rdffr p0.b, p2/z                        # read ffr into p0
        cmpeq p1.b, p0/z, z0.b, 0
        brka p0.b, p0/z, p1.b                   # break after
        st1b z0.b, p0, [x0, x2]
        incp x2, p0.b
        b.none loop
        ret                                     # function exit
```

PREDICATE REGISTERS

```
vadd %Vd, %Vn, %Vm, %Pp
```

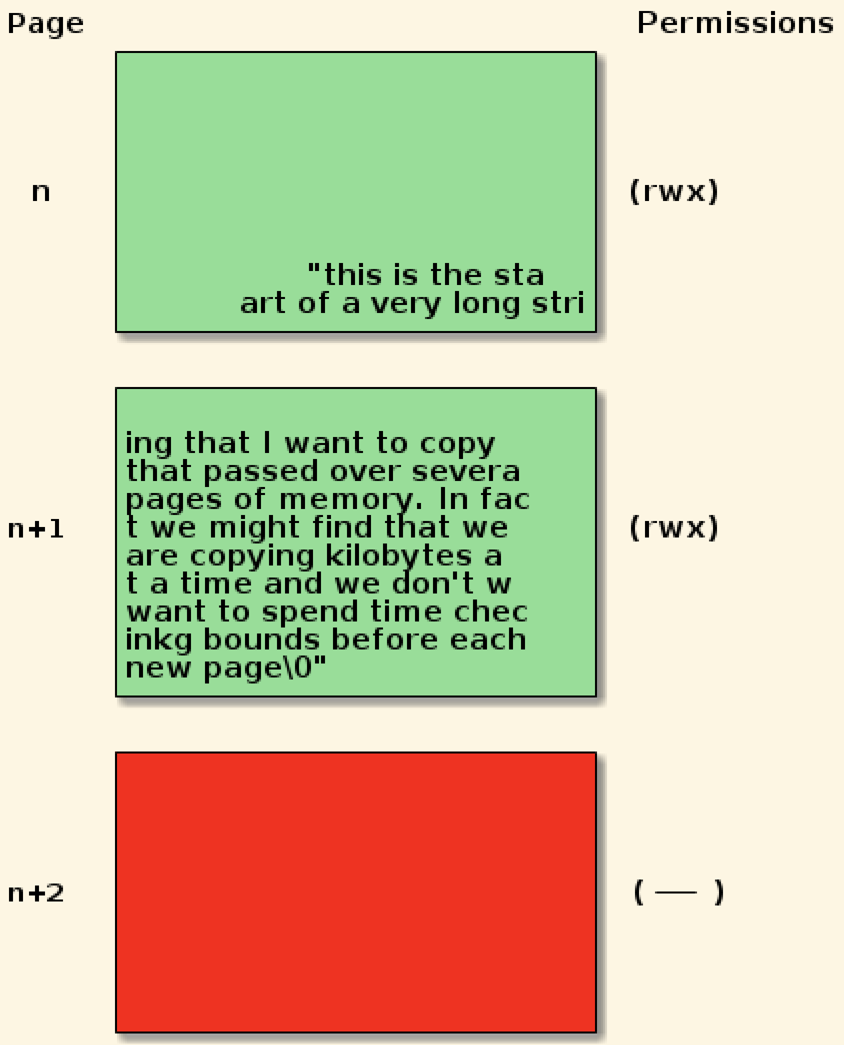


STRCPY (SVE ASSEMBLY SETUP)

```
sve_strcpy:
; setup index and set p2 all true
  mov x2, 0
  ptrue p2.b
```

```
loop:
; clear first fault register, load into z0
  setffr
  ldff1b z0.b, p2/z, [x1, x2]
; did we truncate due to fault?
  rdffr p0.b, p2/z
```

FIRST FAULT REGISTER



STRCPY (SVE ASSEMBLY REST)

```
sve_strcpy:
; setup index and set p2 all true
  mov x2, 0
  ptrue p2.b
loop:
; clear first fault register, load into z0
  setffr
  ldff1b z0.b, p2/z, [x1, x2]
; did we truncate due to fault?
  rdffr p0.b, p2/z
```

```
; any 0's in z0.b
  cmpeq p1.b, p0/z, z0.b, 0
  brka p0.b, p0/z, p1.b
```

```
; store the string to destination
  st1b z0.b, p0, [x0, x2]
```

```
; how many bytes did we copy?
  incp x2, p0.b
```

```
; more?
  b.none loop
  ret
```

RECAP

- Virtualization
  - many flavours
- Vectors
  - large registers
  - growing usage
  - data parallelism

VECTORS MEET (TINY) CODE GENERATION

- QEMU's TCG Mode
- Software only virtualisation

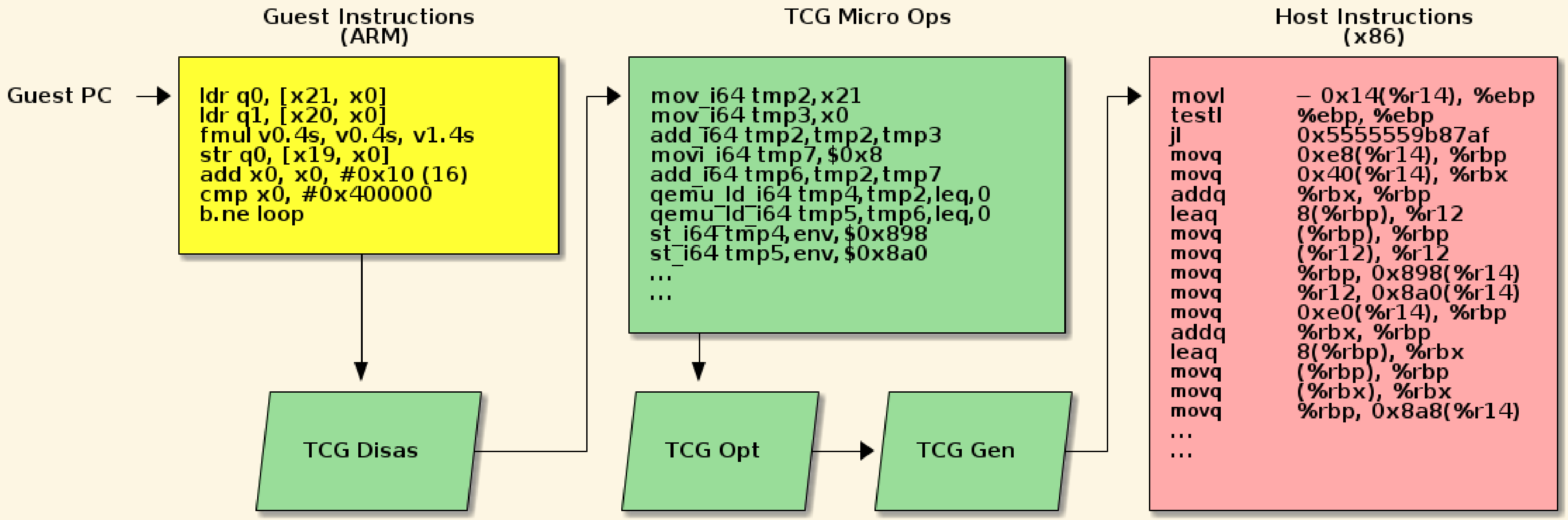
THE X TO Y PROBLEM

- 20 guest architectures
- 7 TCG Backends

WHY CODE GENERATION?

- interpreting slow
- common processor functionality
  - logic
  - arithmetic
  - flow control
- compiler for machine-code

CODE GENERATION



```
ldr q0, [x21, x0]
ldr q1, [x20, x0]
fmul v0.4s, v0.4s, v1.4s
str q0, [x19, x0]
add x0, x0, #0x10 (16)
cmp x0, #0x400000
b.ne loop
```

```
mov_i64 tmp2,x21
mov_i64 tmp3,x0
add_i64 tmp2,tmp2,tmp3
movi_i64 tmp7,$0x8
add_i64 tmp6,tmp2,tmp7
qemu_ld_i64 tmp4,tmp2,leq,0
qemu_ld_i64 tmp5,tmp6,leq,0
st_i64 tmp4,env,$0x898
st_i64 tmp5,env,$0x8a0
...
...
```

```
movl      -0x14(%r14), %ebp
testl     %ebp, %ebp
jl        0x5555559b87af
movq      0xe8(%r14), %rbp
movq      0x40(%r14), %rbx
addq      %rbx, %rbp
leaq      8(%rbp), %r12
movq      (%rbp), %rbp
movq      (%r12), %r12
movq      %rbp, 0x898(%r14)
movq      %r12, 0x8a0(%r14)
movq      0xe0(%r14), %rbp
addq      %rbx, %rbp
leaq      8(%rbp), %rbx
movq      (%rbp), %rbp
movq      (%rbx), %rbx
movq      %rbp, 0x8a8(%r14)
...
...
```

FLOAT MULTIPLY C CODE

```
float *a, *b, *out;
...
for (i = 0; i < SINGLE_OPS; i++)
{
    out[i] = a[i] * b[i];
}
```

FLOAT MULTIPLY: ASSEMBLER BREAKDOWN

```
loop:
```

```
; load data from array
  ldr q0, [x0, x20]
  ldr q1, [x0, x19]
```

```
; actual calculation
  fmul v0.4s, v0.4s, v1.4s
```

```
; save result
  str q0, [x0, x1]
```

```
; loop condition
  add x0, x0, #0x10 (16)
  cmp x0, #0x400000 (4194304)
  b.ne loop
```

TCG IR: LDR Q0, [X0, X21]

Load q0 (128 bit) with value from x21, indexed by x0

```
; calculate offset
  mov_i64 tmp2,x21
  mov_i64 tmp3,x0
  add_i64 tmp2,tmp2,tmp3
```

```
; offset for second load
  movi_i64 tmp7,$0x8
  add_i64 tmp6,tmp2,tmp7
```

```
; load from memory to tmp
  qemu_ld_i64 tmp4,tmp2,leq,0
  qemu_ld_i64 tmp5,tmp6,leq,0
```

```
; store in quad register file
  st_i64 tmp4,env,$0x898
  st_i64 tmp5,env,$0x8a0
```

TCG IR: FMUL V0.4S, V0.4S, V1.4S

```
; get adddress of fpst
 movi_i64 tmp3,$0xb00
 add_i64 tmp2,env,tmp3
```

```
; first fmul.s
 ld_i32 tmp0,env,$0x898
 ld_i32 tmp1,env,$0x8a8
; call helper
 call vfp_muls,$0x0,$1,tmp8,tmp0,tmp1,tmp2
 st_i32 tmp8,env,$0x898
```

```
; remaining 3 fmul.s
 ld_i32 tmp0,env,$0x89c
 ld_i32 tmp1,env,$0x8ac
 call vfp_muls,$0x0,$1,tmp8,tmp0,tmp1,tmp2
 st_i32 tmp8,env,$0x89c
 ...
 ...
```

TCG TYPES

Type	
TCGv_i32	32 bit integer type
TCGv_i64	64 bit integer type
TCGv_ptr*	Host pointer type (e.g. cpu->env)
TCGv*	target_ulong

TCG TYPES AND TGC OPS

- TCGOp has explicit sizes/params

```
tcg_gen_addi_i32(TCGv_i32 ret, TCGv_i32 arg1, int32_t arg2);
tcg_gen_addi_i64(TCGv_i64 ret, TCGv_i64 arg1, int64_t arg2);
```

TYPES FOR VECTORS?

- Type for each Vector Size?
  - TCGv_i128, TCGv_i256…
- Type for each Vector Layout?
  - TCGv_i64x2, TCGv_i32x4…

PROBLEM

Each TCGType -> more TCGOps

TCG_VEC DESIGN PRINCIPLES

- Support multiple vector sizes
- without exploding TCGOp space
- Helpers dominate floating point
- avoid marshalling, pass pointers

TCG_VEC CODE GENERATION

Guest (ARM)

```
eor v0.16b, v0.16b, v1.16b
```

TCG Ops

```
ld_vec tmp8,env,$0x8a0,$0x1
ld_vec tmp9,env,$0x8b0,$0x1
xor_vec tmp10,tmp8,tmp9,$0x1
st_vec tmp10,env,$0x8a0,$0x1
```

Host (x86, SSE)

```
vmovdqu  0x8a0(%r14), %xmm0
vmovdqu  0x8b0(%r14), %xmm1
vpxor    %xmm1, %xmm0, %xmm0
vmovdqu  %xmm0, 0x8a0(%r14)
```

TCG_VEC GIVES US

- better code generation
- more efficient helpers

BENCHMARKS (NSEC/KOP)

Benchmark	Native	TCG	TCG_vec
bytewise-xor	670	331	632
bytewise-xor-stream	235	330	450
wordwide-xor	1349	687	1260
bytewise-bit-fiddle	396	716	**521**
float32-mul	2717	8401	8665

BYTEWISE BIT FIDDLE: C CODE

```c
uint8_t *and, *add, *sub, *xor, *out;
...
for (i = 0; i < BYTE_OPS; i++)
{
    uint8_t value = out[i];
    value |= i & and[i];
    value += add[i];
    value ^= xor[i];
    value -= sub[i];
    out[i] = value;
}
```

BYTEWISE BIT FIDDLE: ASSEMBLY

```
; main loop
  mov x0, #0x0
  mov v1.16b, v29.16b
  add v0.2d, v1.2d, v27.2d
  add v17.2d, v1.2d, v26.2d
  add v2.2d, v1.2d, v25.2d
  add v16.2d, v1.2d, v23.2d
  add v7.2d, v1.2d, v21.2d
  add v20.2d, v1.2d, v24.2d
  xtn v19.2s, v1.2d
  xtn2 v19.4s, v0.2d
  add v18.2d, v1.2d, v22.2d
  ...
  ...
  eor v0.16b, v0.16b, v3.16b
  sub v0.16b, v0.16b, v2.16b
  str q0, [x19, x0]
  add x0, x0, #0x10 (16)
  cmp x0, #0x400000 (4194304)
  b.ne #-0x8c (addr 0x4011a0)
```

BENCHMARKS (NSEC/KOP)

With -funroll-loops

Benchmark	QEMU	QEMU TCG_vec
bytewise-xor	332	338
bytewise-xor-stream	169	185
wordwide-xor	670	**631**
bytewise-bit-fiddle	661	**469**
float32-mul	7941	7634

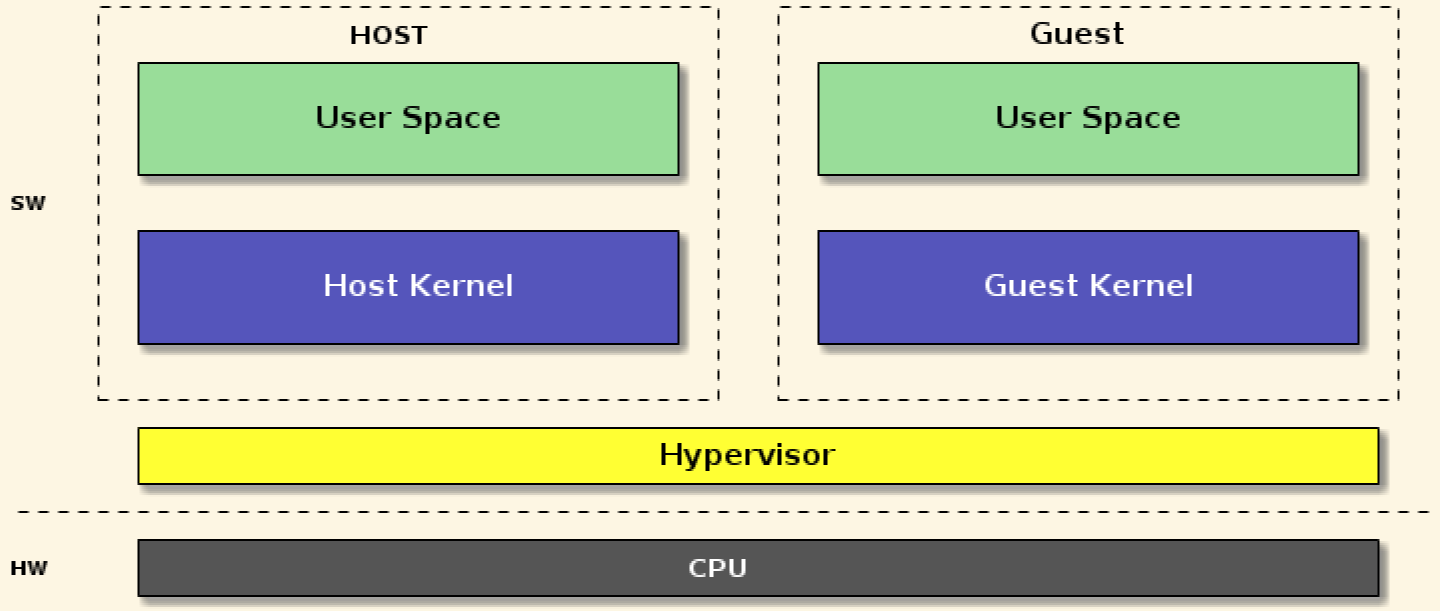
FURTHER WORK

- ld/st handling
- better register liveliness

VECTORS MEET KVM*

- Xen
- HAXM (Windows)
- HVM (MacOS)

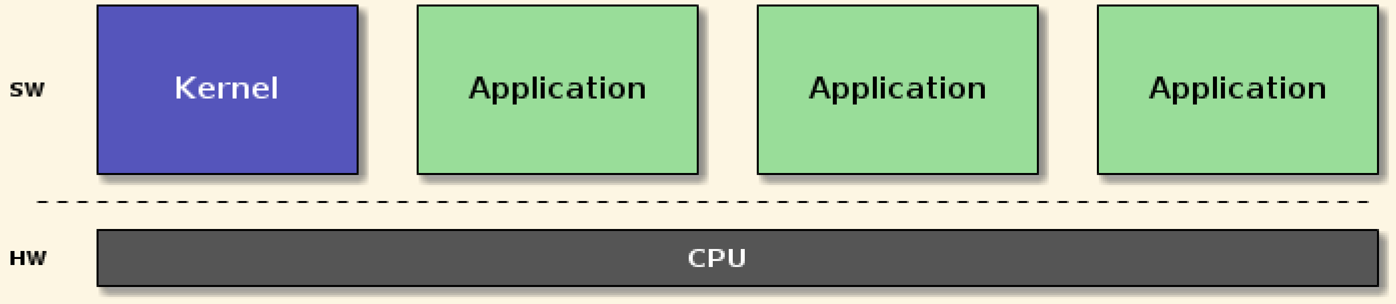
ARCHITECTURE



CPU RESOURCES

- Shared execution environment
- Virtualized resources for guest
    - Trap and Emulate
    - Context Switch

SWAPPING CONTEXT IN HOST KERNEL



SIZE OF ARMV8 CONTEXTS

- 32 x 64 bit integer regs (256 bytes)
- 32 x 2048 bit SVE regs (8192 bytes)
- 32 times bigger!

WHO USES SIMD (AND FP!)

- Userspace
  - dedicated vectorized workloads
  - accelerated library functions
- Kernel
  - Crypto
  - RAID
- Hypervisor
  - Not really

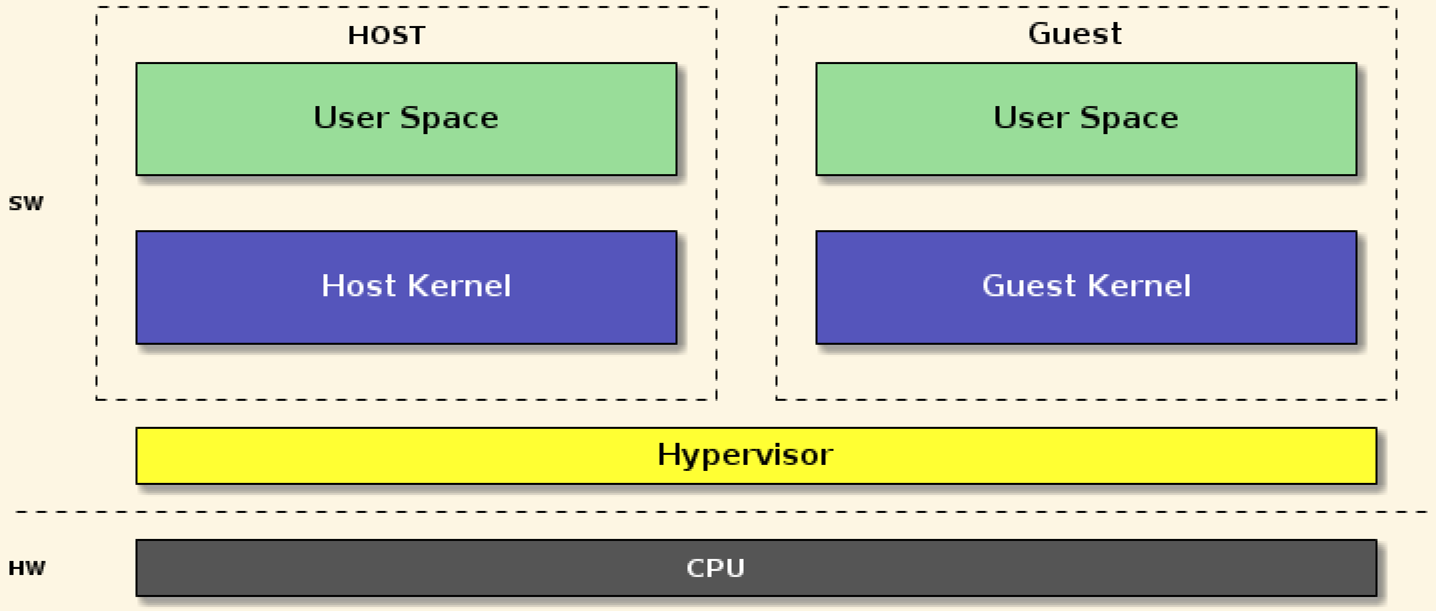
DETECTING USAGE

- Disable SIMD/FPU access
- First usage with Trap
  - swap context
  - enable SIMD/FPU
  - return to trapped insn

DEFERRED STATE BOOKEEPING

- per CPU variable
  - fpsimd_last_state
- per Task Variables (task_struct)
  - fpsimd_state
  - TIF_FOREIGN_FPSTATE flag

VM IS MOSTLY THE SAME



ENABLING SVE ON ARM

- Kernel support in 4.15
- Enabling SVE for KVM guest
  - work in progress

SUMMARY

- Vectors are great
- Vectors are large!
- Need special handling by
  - Kernels
  - Hypervisors
  - Emulators

QUESTIONS?

EXTRA SLIDES

BENCHMARK CODE

See:
https://github.com/stsquad/testcases/blob/master/aarch64/vector-benchmark.c