arm

Upcoming Arm architectural features

Andrew Wafaa <andrew.wafaa@arm.com>

Director Open Source, Communities

FreeBSD Vendor Summit – October 2019

Introduction

Many additions to the Arm architecture have been made since ARMv8.0 was released. In this talk, I aim to talk through the some of the more interesting ones as they can have ramifications in software support.

I will go through a lot of features quickly, in order to show:

1. That they exist,
2. Give some indication on the progress of their support in software (primarily in Linux).

These slides are a **rough** indication of architectural support present/coming. I have links to documentation which I strongly suggest interested parties to read. ☺

I am a mouth piece for our engineers, so please validate all answers I give!

Architectural features

In this talk I aim to cover...

- ARMv8.1
  - Large System Extensions
- ARMv8.2
  - 52-bit address spaces
  - Persistent memory support
  - Scalable Vector Extension
  - Statistical profiling extension
  - RAS
- ARMv8.3
  - Nested virtualisation
  - Pointer authentication
- ARMv8.4
  - Enhanced Nested virtualisation
  - MPAM
- ARMv8.5
  - Deep persistence
  - Memory Tagging Extension
  - Mitigations against side-channel attacks (retroactively added to Arm V8.0)
- New Technologies for the Arm A-Profile Architecture
  - Transactional Memory Extension
  - SVE2

arm

Large System Extensions – ARMv8.1-LSE

Let's consider: `__atomic_fetch_add(&var, 1, __ATOMIC_RELAXED);`

```
beg:ldxr    w1, [x0]
    add     w1, w1, #0x1
    stxr    w2, w1, [x0]
    cbnz    w2, beg
```

`-march=armv8-a+lse`

```
stadd    w1, [x0]
```

- LSE allows atomic operations (CAS, SWP, LD<OP>, ST<OP>, where OP is one of ADD, CLR, EOR, SET, SMAX, SMIN, UMAX, UMIN) in a single instruction,
- This also allows the system to perform the atomic operations outside the CPU,
- Unfortunately, as atomic instructions are rather fundamental it does not make sense to select them with HWCAPs in IFUNCs,
- There is a multi-lib mechanism in glibc 2.28 that supports `HWCAP_ATOMIC`, however that would provoke multiple binaries, thus multiple builds,
- A better way (for distros) is being worked on...

arm

Another way to deploy LSE `-matomic-ool`

- There is an "out-of-line" atomics patch set for gcc,
- Our test case then compiles to:

```
0000000000400720 <__aa64_stadd4_relax>:
400720:       b0000082        adrp    x2, 411000 <getauxval@GLIBC_2.17>
400724:       3940e442        ldrb    w2, [x2, #57]
400728:       350000c2        cbnz    w2, 400740 <__aa64_stadd4_relax+0x20>
40072c:       885f7c22        ldxr    w2, [x1]
400730:       0b000042        add     w2, w2, w0
400734:       88027c22        stxr    w2, w2, [x1]
400738:       35fffa2         cbnz    w2, 40072c <__aa64_stadd4_relax+0xc>
40073c:       d65f03c0        ret
400740:       b820003f        stadd   w0, [x1]
400744:       d65f03c0        ret
```

- We perform a **direct branch** to select the appropriate code sequence at run time,
- Practical tests on DPDK lock stress test showed very little perf degradation,
- Hopefully, this patch set will make it into gcc soon.

52-bit address spaces – ARMv8.2-LVA, ARMv8.2-LPA

When running with a 64KB PAGE_SIZE it is possible, on supported hardware, to have a 52-bit physical address (PA), intermediate physical address (IPA) and virtual address (VA). (Normally the maximum size is 48-bit).

- 52-bit PA support landed in Linux 4.16,
- 52-bit IPA support landed in 4.20,
- 52-bit VAs for userspace landed in 5.0,
- 52-bit VAs for kernel space are queued up and, hopefully, will land in 5.4,

Just like with x86, larger user space VAs on Arm are achieved by supplying a "high" hint to mmap.

Currently a 4KB PAGE_SIZE does not have support for an address space larger than 48-bit.

Persistent memory support in Arm

DC CVAP & DC CVADP

- ARMv8.2-DCPoP introduced the notion of cleaning to Point of Persistence (PoP),
  - Writes to the point of persistence are maintained in the event of power loss,
- ARMv8.2-DCCVADP introduced the Point of Deep Persistence (PoDP),
  - Writes to the point of deep persistence are maintained in the event of an instantaneous power loss.

In the Linux kernel the following is being worked on:

- Memory hot-add/remove support,
- PTE_DEVMAP,

With some miscellaneous cleanup patches this then allows ZONE_DEVICE and DAX.

In order for the arm64 kernel to pick up NVDIMMs, the firmware needs to expose the requisite NFIT information in ACPI.

On the userspace side, PMDK is being investigated.

Scalable Vector Extension (SVE) – Example!

My favourite Arm instruction – `ldff1b`

In this example we have a block of memory pointed to by x0. However the memory block crosses a page boundary and the next page isn't present so would cause a fault on access.

Memory at `x0`	0x01	0x02	0x03	0x04	0x05	invalid	invalid	invalid	…
Register `p1`	1	1	0	0	1	1	1	0	…
We then perform a: `LDFF1B Z0.B, P1/Z, [x0]`									
Result in `Z0`	0x01	0x01	0x00	0x00	0x05	0x00	0x00	0x00	0x00

Register `p1` is a *predicate register*, it dictates which lanes are affected by an operation. `P1/Z` is a *zero-ing predication* meaning "set lanes to 0 that aren't selected".

The result is stored in register `z0` which is an SVE register. The number of lanes is IMPLEMENTATION DEFINED (but discoverable at run time).

The instruction above won't provoke a page fault if it loads the first lane successfully. ☺

arm

Scalable Vector Extension (SVE)

An **optional** feature in the ARMv8.2 architecture

- SVE adds another set of registers Z0-Z31 (lower 128 bits of map to V0-V31), size is implementation defined (it can vary from 128b to 2048b),
- Predicate registers P0-15 are also added, P0-P7 can be used to affect how an instruction operates (they are “governing predicates”),
- In addition to predication, we have the First Fault Register (FFR), this is used to handle crossing memory page boundaries,
- All the above allows one to program SVE without knowing how big the vectors Z0-Z31 are, or even how big the array of data is at compile time,
- In other words, there are significantly more scenarios where SVE can be applied where Neon would be unsuitable, for instance, string handling!
- Ideally SVE can be covered in a talk/course of its own…

SVE example - `strlen`

Example stolen from https://alastairreid.github.io/papers/sve-ieee-micro-2017.pdf

1. We read data with `ldff1b`, from `x1` into `z0`. Successful reads then moved into `p1`,
2. Successfully read elements compared to 0. Equalities stored in `p2`, and then "inverted" by `brkbs` - break before first true condition,
3. `x1` is then advanced by `p2`,
4. If any 0s found return `x1 - x0`, otherwise loop.

```
// ------------------------------------
//      int strlen(const char *s) {
//          const char *e = s;
//          while (*e) e++;
//          return e - s;
//      }
// ------------------------------------
// x0 = s

// Unoptimized SVE strlen
strlen:
     mov      x1, x0                  // e=s
     ptrue    p0.b                    // p0=true
.loop:
     setffr                           // ffr=true
     ldff1b   z0.b, p0/z, [x1]        // p0:z0=ldff(e)
     rdffr    p1.b, p0/z              // p0:p1=ffr
     cmpeq    p2.b, p1/z, z0.b, #0    // p1:p2=(*e==0)
     brkbs    p2.b, p1/z, p2.b        // p1:p2=until(*e==0)
     incp     x1, p2.b                // e+=popcnt(p2)
     b.last   .loop                   // last active=>!break
     sub      x0, x1, x0              // return e-s
     ret
```


arm

Software support for SVE

- SVE is supported in gcc 8.1 & llvm 7,
  - Auto vectorisation is expected to improve with subsequent releases,
- The Linux kernel supports SVE from 4.15,
- (SVE2 support is relayed to userspace from Linux 5.2 and targeted for gcc 10 & llvm 9),
- QEMU supports SVE from version 3.1,
- Arm also has an SVE instruction emulator based on Dynamo-Rio: https://developer.arm.com/tools-and-software/server-and-hpc/arm-architecture-tools/arm-instruction-emulator
- … and some commercial C/C++/Fortran compilers…

Statistical profiling extension

An optional feature in the ARMv8.2 architecture

- Traditional sampling with a PMU
  - Relies on interrupts, then the kernel to gather data,
- Statistical profiling extension
  - Gives contextual information for the event,
  - Can sample everywhere (allowed by security),
- SPE is accessed in perf, as an example:

```
perf record -e
arm_spe/ts_enable=1,pa_enabl
e=1/
```

- Perf supported landed in Linux 4.16,
- Support for ACPI SPE landed in Linux 5.3.

```
... ARM SPE data: size 2097152 bytes
LD
VA 0xffff00000f293bc0
PA 0xfb24ebc0 ns=1
LAT 0 XLAT
EV RETIRED L1D-ACCESS TLB-ACCESS
PC 0xff00000815c400 el3 ns=1
LAT 0 TOT
TS 39395093558
```


arm

Nested virtualisation – ARMv8.3-NV and ARMv8.4-NV

Unfortunately virtualisation support in ARMv8 did not originally include nesting. Thus on older hardware it is not possible to run a guest within a guest...

Nested virtualisation was introduced in ARMv8.3-NV and provided a means to run a hypervisor in EL1 (with trapping to the host hypervisor to handle scenarios). This was enhanced in ARMv8.4-NV to replace a lot of the trapping with an array lookup.

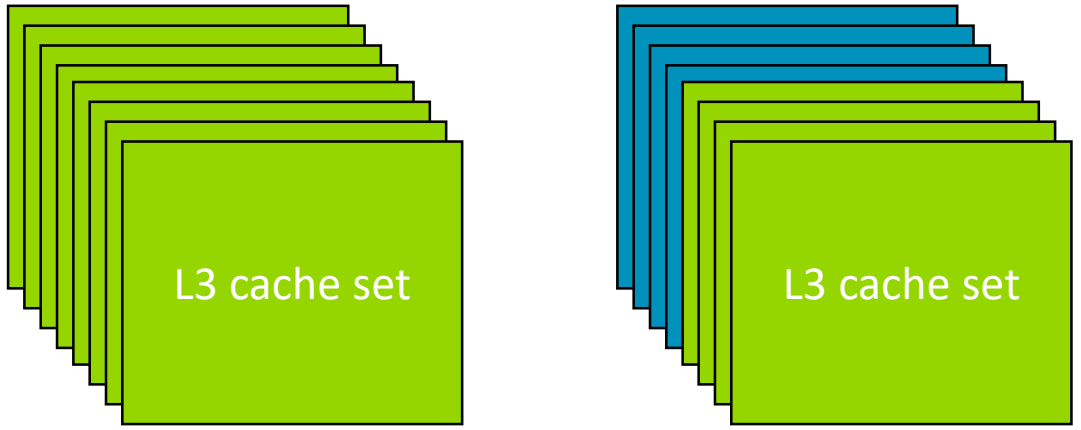
Nested virtualisation is an optional feature.

Nested virtualisation support in the kernel/KVM is being developed and some patches to enable ARMv8.3-NV have been posted to lakml.

Memory Partitioning and Monitoring (MPAM)

Introduced as optional in ARMv8.4

- MPAM allows one to partition the L3 cache s.t. sets are reserved,
- This can then provide more predictable performance to certain tasks,
- MPAM also allows monitoring of memory traffic,
  - i.e. one can monitor task memory usage.
- MPAM support is still being developed for the Linux kernel.

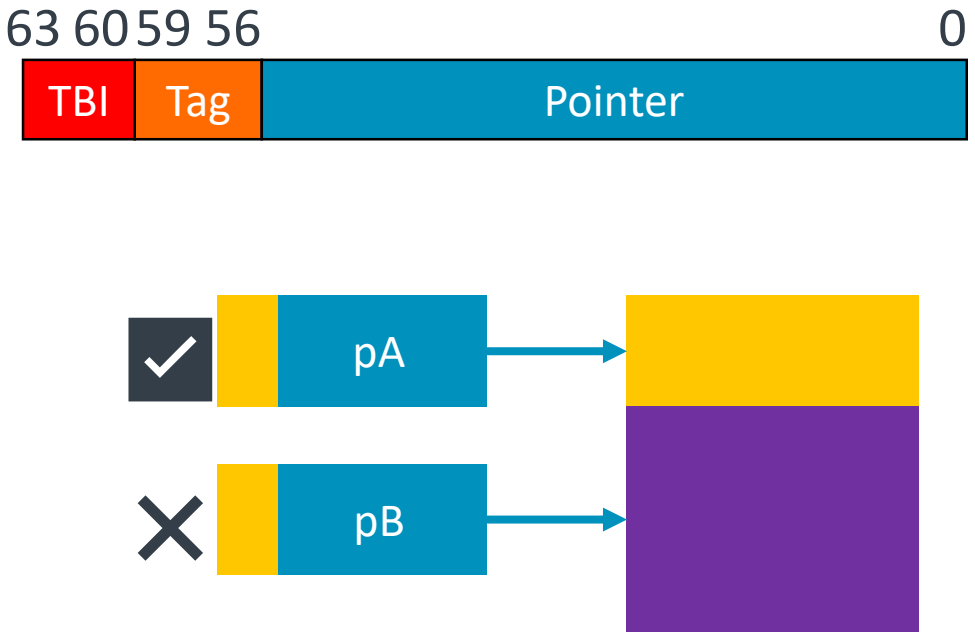


Half of cache reserved for process

arm

Memory Tagging Extension – ARMv8.5-MemTag

- Introduced in ARMv8.5 to protect against buffer overrun and use-after-free,
- VA[59:55] are a logical tag (or colour),
- A physical address tag is assigned with 16 byte granularity,
- The physical address tag is stored in memory,
- The logical and physical address tags are *Tag Checked* and an exception *can* be thrown on mismatch.



arm

Employing Memory Tagging Exception

This is a subset of instructions for MTE…

- Generating a random tag:
  `IRG x1, x0` – generate random tag, apply it to address in x0, store result in x1
- Generating two more different random tags – continuing from above:
  `GMI x2, x1, x2` – x2 contains "excluded set" so far only tag from x1
  `IRG x3, x0, x2` – x3 contains random tag applied to x0, it cannot match x1
  `GMI x2, x3, x2` – x2 contains "excluded set", so far from x1 and x3
  `IRG x4, x0, x2` – x4 contains random tag applied to x0, it cannot match x1 or x3
  (if we run out of random tags, IRG will store 0 in x2 to notify us)
- Storing a tag to physical memory:
  `STG x1, [x0]` – extract tag from x1, store in physical memory pointed to by x0
- Loading a tag from physical memory:
  `LDG x2, [x0]` – read tag from x0 store in x2

Tag Checking

There are three kinds of Tag Checking:

1. *No Effect*,
2. *Synchronous Exception failure*. A Tag Check failure results in a data abort being raised (if the memory access was a write it will be blocked). Userspace sees a signal with the context at the point of failure. (i.e. similar to a SIGSEGV on invalid pointer access),
3. *Asynchronously Accumulated*. The memory access takes place, and any Tag Check failures cause bit 0 to be set in `TFSRE0_EL1`.

Synchronous Exception failure is more expensive than Asynchronously Accumulated but gives extra security.

Mitigations against speculation/side-channel attacks

I would **strongly** recommend reading the references with several coffees to take these in…

The following are **optional** in ARMv8.0 and **mandatory** in ARMv8.5.

- ARMv8.0-SB, Armv8.0 Speculation Barrier
  - Introduces Consumption of Speculative Data Barrier (CSDB),
  - The CSDB instruction is a memory barrier instruction that controls speculative execution and data value prediction
- ARMv8.0-SSBS, Armv8.0 Speculative Store Bypass Safe
  - Introduces Speculative Store Bypass Barrier (SSBB),
  - The SSBB is a memory barrier that prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions.
- ARMv8.0-PredInv, Armv8.0 Prediction Invalidation
  - ARMv8.0-PredInv adds the CFP RCTX, CPP RCTX, DVP RCTX, CFPRCTX, CPPRCTX, and DVPRCTX System instructions. These instructions prevent predictions based on information gathered from earlier execution within a particular execution context from affecting the later speculative execution within that context, to the extent that the speculative execution is observable through side channels.

More speculation

The following are **mandatory** in both ARMv8.0 and ARMv8.5 and advertise the susceptibility of the CPU to cache speculation variants:

- ARMv8.0-CSV2, Armv8.0 Cache Speculation Variant 2
- ARMv8.0-CSV3, Armv8.0 Cache Speculation Variant 3

The above provide system registers that are available in all Arm implementations.

Arm Transactional Memory Extension (TME)

See references section for instruction encodings and pseudocode…

Programmer writes

```
tstart x0
cbnz x0, fallback
// transactional code here
tcommit
```

Hardware provides

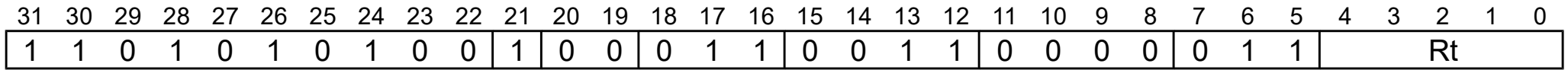
- Strong isolation
  - Non-interference & containment from both transactional and non-transactional code.
- Failure atomicity
  - Architectural changes discarded on failure.
  - All instructions commit or none.
- Best-effort transactions
  - No forward progress guarantee, SW must provide non-transactional fallback path.
  - Good for multi-client “server” applications with large, rarely contended data structures.

An example of **public** documentation for New Technologies

TSTART

This instruction starts a new transaction. If the transaction started successfully, the destination register is set to zero. If the transaction failed or was canceled, then all state modifications that were performed transactionally are discarded and the destination register is written with a non-zero value that encodes the cause of the failure.

System
(TME)



System

```
TSTART <Xt>

if !HaveTME() then UNDEFINED;
integer t = UInt(Rt);
```

Assembler Symbols

<Xt>	Is the 64-bit name of the general-purpose destination register, encoded in the "Rt" field.

arm

References

- ArmARM: https://static.docs.arm.com/ddi0487/ea/DDI0487E_a_armv8_arm.pdf
- Speculation attack mitigation: https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/latest-updates/cache-speculation-issues-update
- ArmARM v8-A Supplement - MPAM: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0598a.b/index.html
- ArmARM v8-A Supplement - SVE: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0584a.f/index.html
- ArmARM v8-A Supplement - RAS: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0587c.b/index.html
- https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/new-technologies-for-the-arm-a-profile-architecture

References (2)

- https://static.sched.com/hosted_files/bkk19/3c/BKK19-202_New-Technologies-in-Arm-Architecture.pdf
- https://developer.arm.com/architectures/cpu-architecture/a-profile/exploration-tools
- https://static.docs.arm.com/ddi0602/b/ISA_A64_xml_futureA-2019-06_OPT.pdf

arm

Thank you for your attention!

Thank You
Danke
Merci
谢谢
ありがとう
Gracias
Kiitos
감사합니다
धन्यवाद
شكرًا
תודה