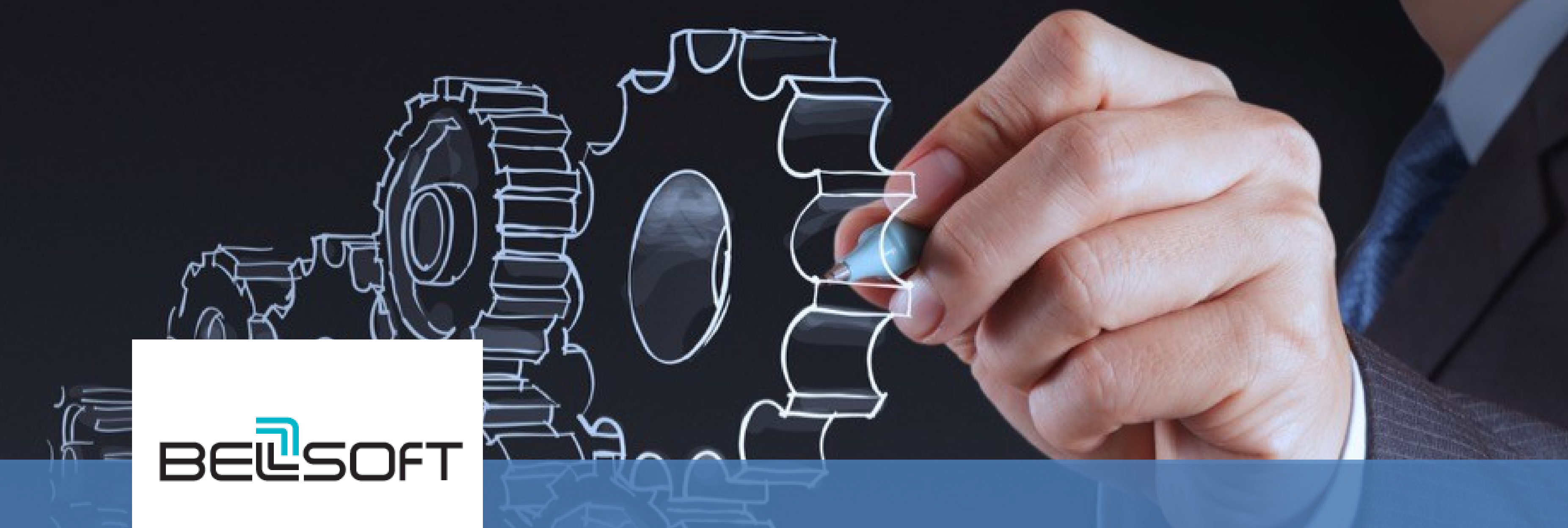


Java on ARM
Theory, Applications and Workloads

Dmitry Chuyko
JVM Team

Who we are

Dmitry Chuyko

@dchuyko

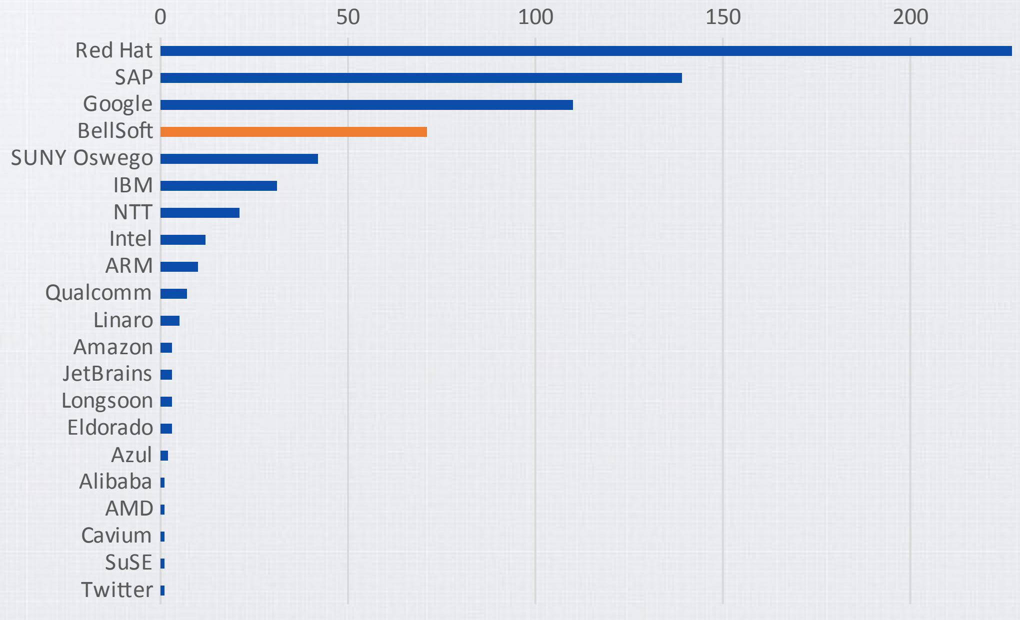
Liberica JDK – verified OpenJDK binary
http://bell-sw.com

Ex-employers

Committed to freedom

Liberica
– supported OpenJDK
binaries

http://bell-sw.com

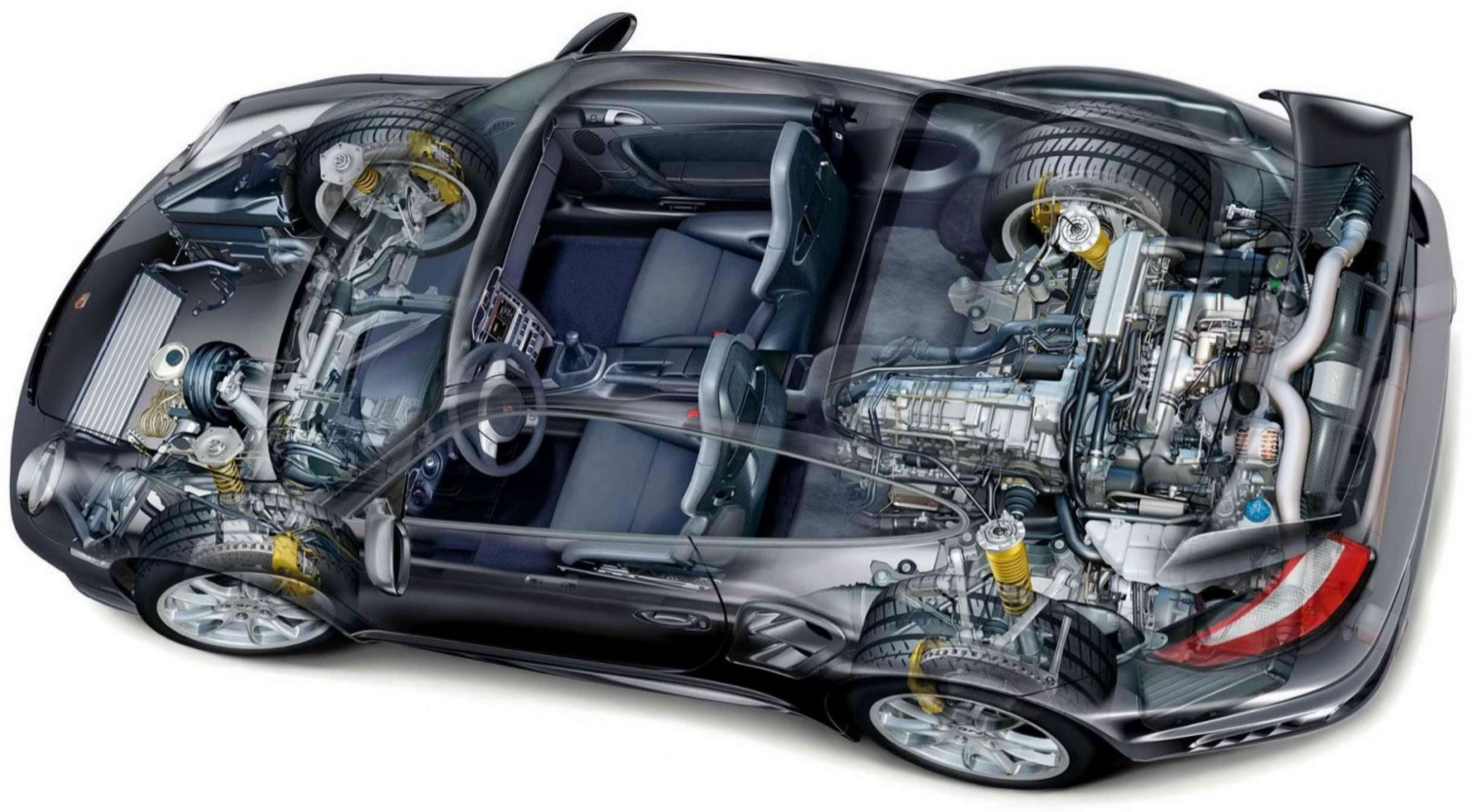


What do we know about Arm?

- Arm = Advanced RISC Machine/Acorn RISC Machine
- Founded in 1985
- UK, Cambridge
- ARM is a RISC architecture

- 30 billion processors shipped in 2013
- Plans to ship 100 billion processors by 2020

11:59
Mon, February 24
Now playing
"AMERICAN MOUTH"
by The Rock Orchestra
14:59 PM
1 new message
Search
Text size



IoT Gateways

Liberica JDK

SuperMicro

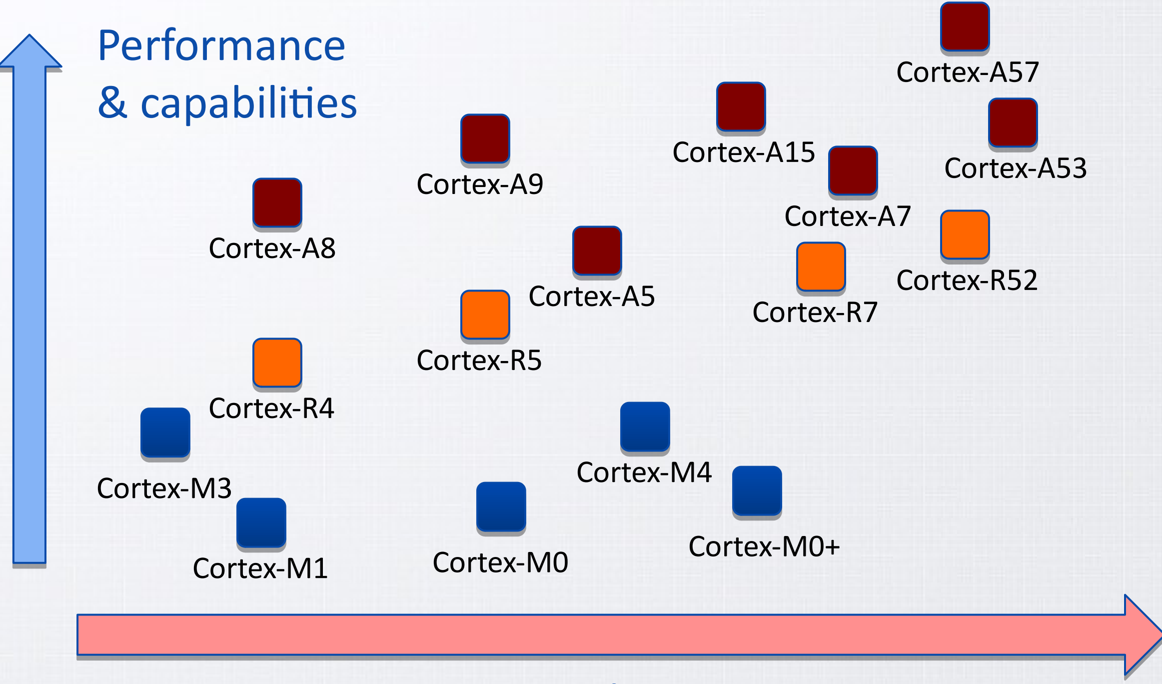
Dell

Eurotech

Advantech

But Servers?

Arm: architecture, profile, implementation



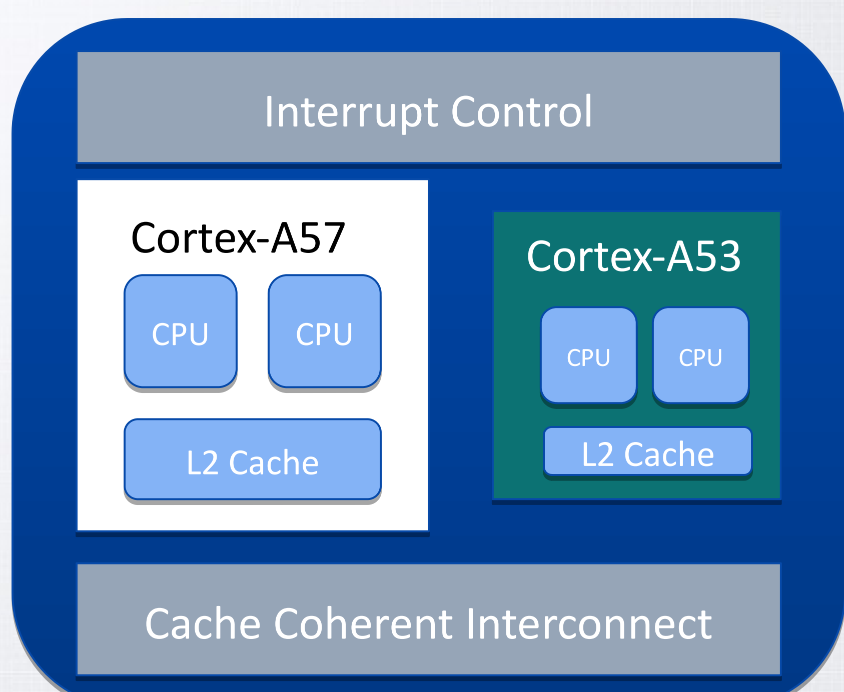
- ARM v7
- Architecture profiles
  - v7-M (Embedded)
  - V7-R (Real-Time)
  - V7-A (Application)

- ARM v8
- Architecture profiles
  - v8-M (Embedded)
  - V8-R (Real-Time)
  - V8-A (Application)

Arm: big.LITTLE

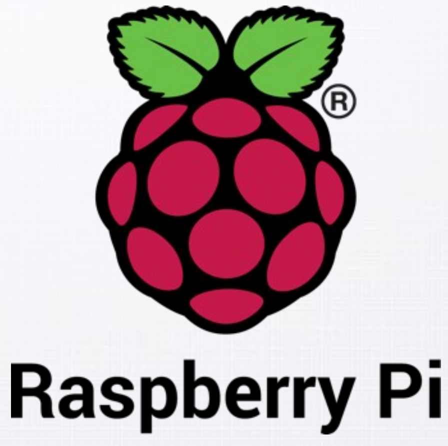
BIG

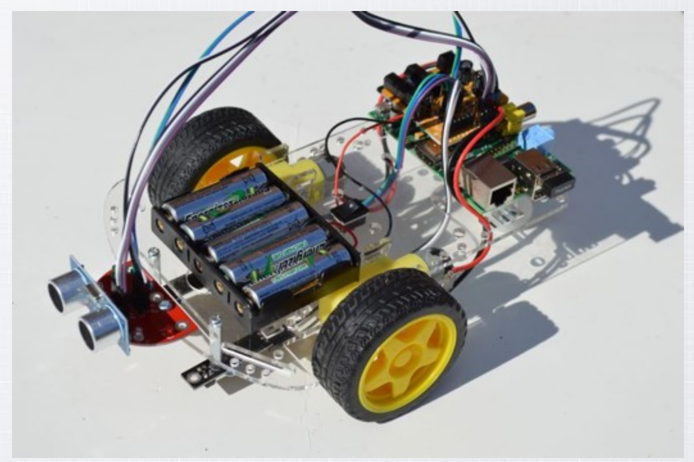
Performance
on-demand



Always
connected

DIY





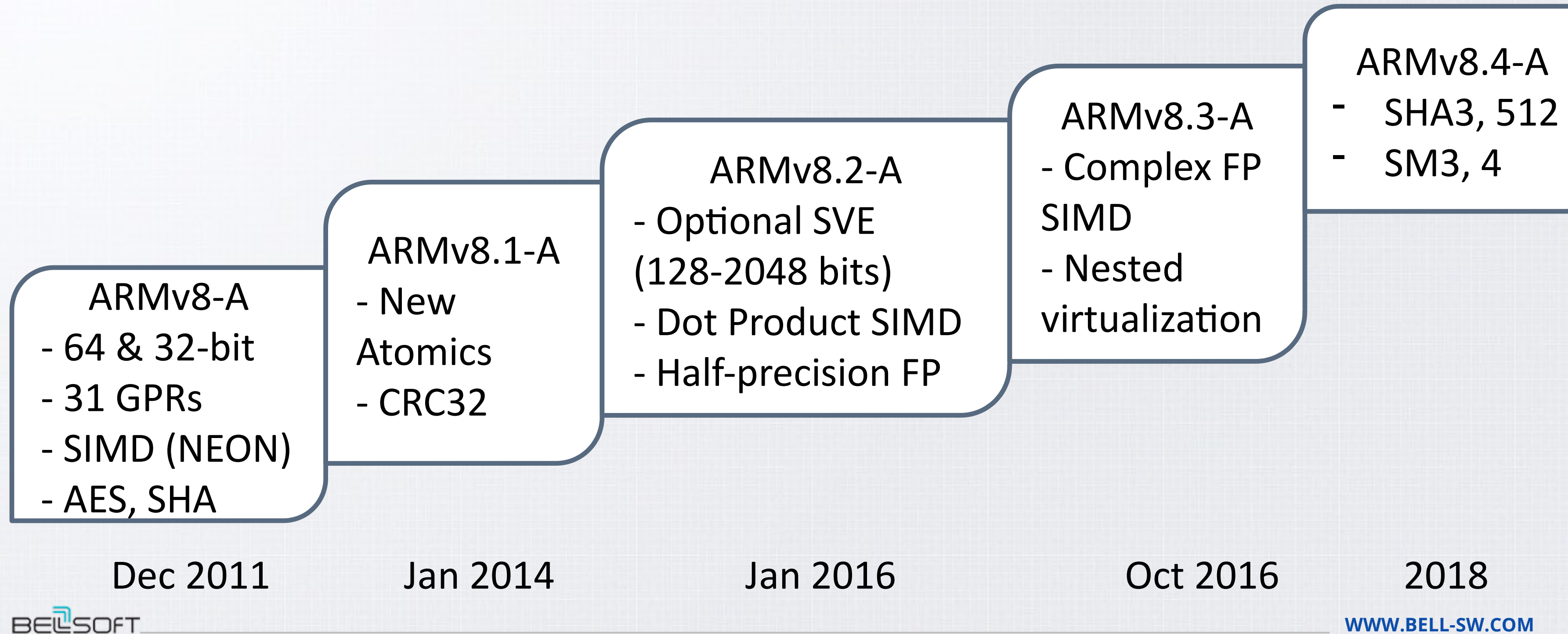
OpenJDK Arm32 port

- Available since OpenJDK 9
  - Minimal VM, Client VM, Server VM
- Works on the Raspberry Pi
- jlink + jdeps
  - Allows to create a smaller runtime (as small as 16 Mb)
- Java FX Embedded
  - Allows to build fancy UI for the Raspberry Pi
    - EGL/DFB acceleration
    - Touch screen support

Minimal VM

- Optimized for footprint, rather than functionality
  - Serial GC
  - C1 JIT compiler
  - No JDWP support
  - No JMX support
- But… it is **< 4 Mb!**
  - Linux x86_64 Server VM: 23 Mb
- jlink @since jdk9
  - java.base with Minimal VM under 16 Mb!
  - Modules for jetty: under 32 Mb

ARMv8-A Specification



Arm architecture licensees

HUAWEI

Neoverse Infrastructure IP Roadmap

Each generation brings faster performance and new infrastructure specific features

16nm	7nm	7nm+	5nm
Cosmos Platform (A72, A75)	Ares Platform	Zeus Platform	Poseidon Platform
Today	2019	2020	2021

~30% per Gen Faster Performance & New Features

arm NEOVERSE

Materials in this presentation are under strict embargo until October 16, 2018 at 11:30 a.m. PT / 2:30 p.m. ET / 7:30 p.m. BST

arm © 2018 Arm Limited

Ampere Computing (ex APM)

32 Arm v8 cores up to 3.3 GHz with Turbo

L 1I | Arm v8 CPU | L1D | L2 Cache

Shared 32 MB L3 Cache

8 x DDR4-2667

72b DDR4 with ECC

Coherent Network

I/O Network

x16 | x16 | x 8 | x1 | x1

PCIe 3.0

42 Lanes of PCIe 3.0

4x | 2x

SATA 3.0 | USB 2.0

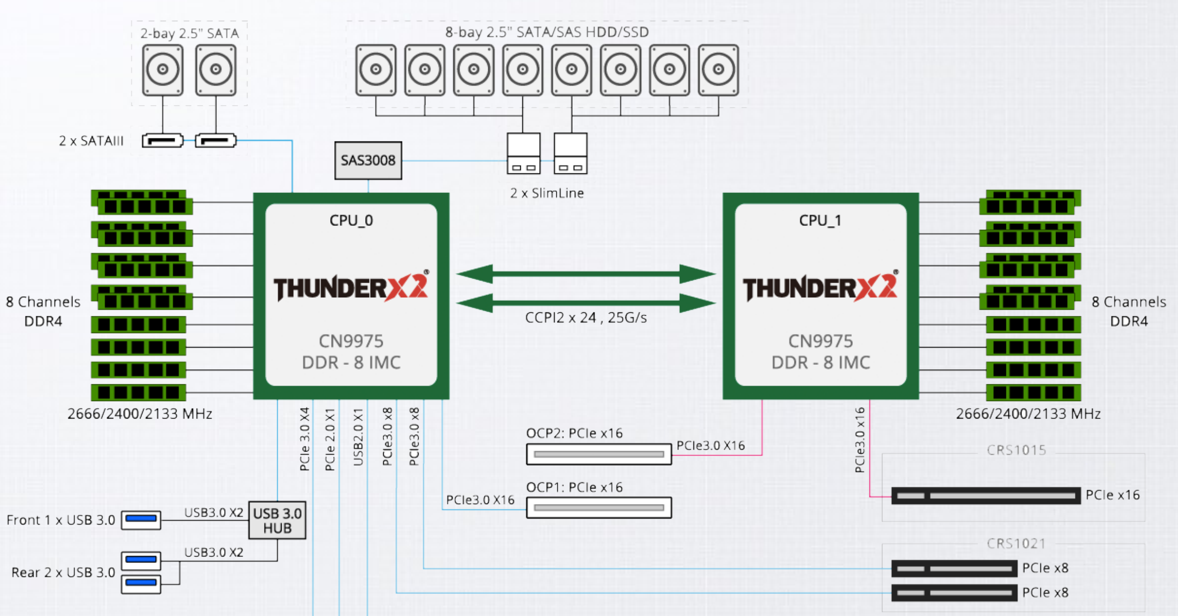
2x Instruction Trace

PMPro | SMPro

Secure Boot and Management Processors Low-Speed Interfaces

Up to 32 cores
Up to 32 threads
8 DDR Channels
32 Mb L3

Cavium/Marvell ThunderX2



32 cores/128 threads
32 Mb L3
8 DDR Channels/socket
Multi-socket
Up to 4 TB RAM

That thing
is real!
MARVE
TCK
VERIFIED
BELLSOFT

Wait, how many threads?

```
Mem[|||||||||||||||||||||||||||          13.1G/511G]
Swp[                                          0K/0K]
Tasks: 50, 660 thr; 244 running
Load average: 205.40 100.67 39.63
Uptime: 30 days, 17:53:24
```

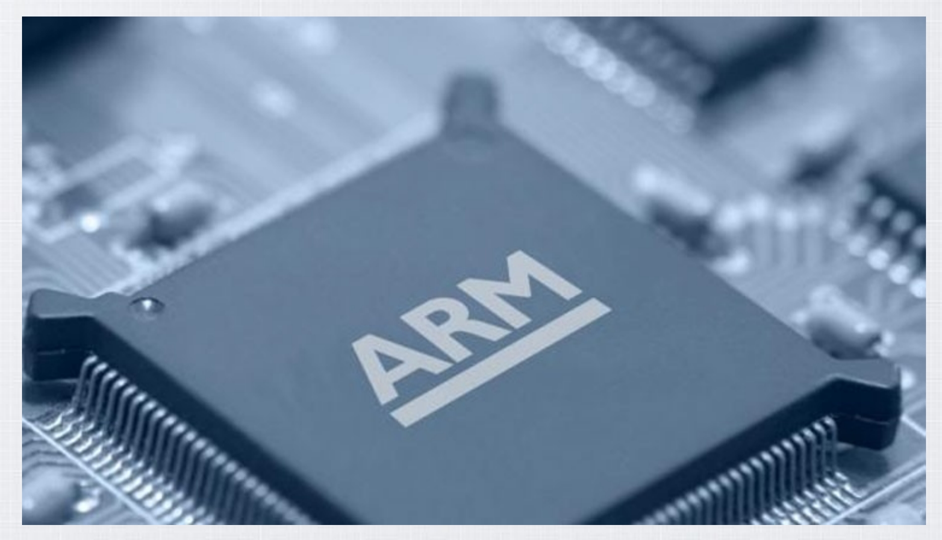
Arm Software ecosystem

Check out if it works on Arm: https://worksonarm.com

Cross-build Docker images

OpenJDK ARM ports

- ARM (32 bit & 64 bit)
  - Full Java SE Spec
  - ARM v6/v7/v8
  - C1 & C2
- AARCH64 (64 bit only)
  - Full Java SE Spec
  - C1 & C2
  - G1 / Parallel GC / Shenandoah (and ZGC is coming)
  - AppCDS, JFR, NMT, AOT



Intrinsics

Intrinsic:

“function (subroutine) available for use in a given programming language which implementation is handled specially by the compiler.”

Intrinsics

- GCC/LLVM
  - Specialized instructions not expressible through regular language constructs
  - Wrappers around libc calls
- HotSpot
  - Manipulation on C2 IR
    - Typically a specialized assembly instruction call for a given architecture
  - Stub – assembler or native routines

What will C2 do with math Java code?

java.lang.Math:

```
/**
 * Returns as a {@code long} the most significant 64 bits of the
 * 128-bit product of two 64-bit factors.
 * @since 9
 */
public static long multiplyHigh(long x, long y) {
   if (x < 0 || y < 0) {
       long x1 = x >> 32;
       long x2 = x & 0xFFFFFFFFL;
       long y1 = y >> 32;
       long y2 = y & 0xFFFFFFFFL;
       long z2 = x2 * y2;
       long t = x1 * y2 + (z2 >>> 32);
       long z1 = t & 0xFFFFFFFFL;
       long z0 = t >> 32;
       z1 += x2 * y1;
       return x1 * y1 + z0 + (z1 >> 32);
   } else { ...
```

What will C2 do with math Java code?


```
java.lang.Math:
/**
 * Returns as a {@code long} the most significant 64 bits of the 128-bit
 * product of two 64-bit factors.
 * @since 9
 */
public static long multiplyHigh(long x, long y) {
    // Use technique from section 8-2 of Henry S. Warren, Jr.,
    // Hacker's Delight (2nd ed.) (Addison Wesley, 2013), 173-174.
    ...
    // Use Karatsuba technique with two base 2^32 digits.
    ...
    return ...;
}
```

Math code in assembly

```
lsr x14, x2, #32
lsr x15, x1, #32
add x11, x14, x11
add x10, x15, x10
mul x14, x15, x14
mul x10, x10, x11
sub x10, x10, x14
```

14 operations with latency 1

Can we make it faster?

- Rewrite as a C + JNI call
  - Well, it will be slower
- Tune HotSpot to optimize IR for this code better*
  - Even if this is possible, this might lead to regressions
- Tune HotSpot to detect this method and substitute optimal code instead

SMULH *Xd, Xn, Xm* *(cost: 4)*

“Signed multiply high”

C2 Intrinsic How-to

1) Add SMULH instruction into ${arch}/assembler_${arch}.hpp
2) Describe a node with this instruction and its cost in ${arch}.ad
3) Mark this method as intrinsic in share/classfile/vmSymbols.hpp
4) Substitute the method with the node

```
bool LibraryCallKit::inline_math_multiplyHigh() {
 set_result(_gvn.transform(new MulHiLNode(arg (0), arg (2))));
 return true;
 }
```

5) Annotate j.l.Math.multiplyHigh() @HotSpotIntrinsicCandidate
6) Measure performance

Benchmarking (throughput)


```
public class MultiplyHighJMHBench {

    @Benchmark
    @OperationsPerInvocation(10000)
    public long bench() {
        long op = System.currentTimeMillis();
        long accum = 0;
        for (int i = 0; i < 10000; i++) {
            accum += Math.multiplyHigh(op + i, op + i);
        }
        return accum;
    }
}
```

SMULH cost: 4

Good for JDK 11!

Let's do something useful for enterprise apps

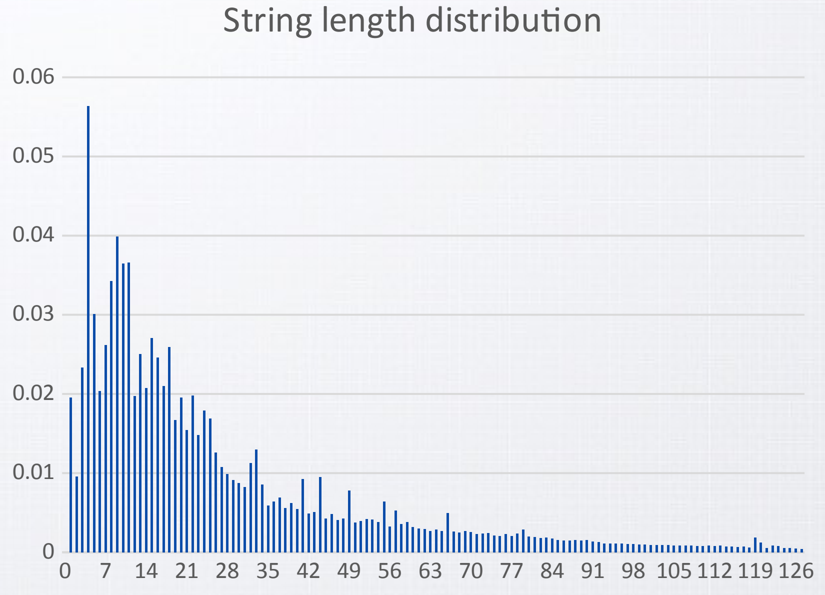
- What does a JVM do when executing a typical enterprise program?
  - Creates, copies objects, strings, arrays, frees memory
  - Searches and compares objects, strings, arrays
  - Checks that the right information is received

String s = new String("Can this work faster?");

- Compact Strings @since JDK 9
  - Most strings do not require UTF-16 as inner representation
  - Inner representation of strings:
    - char[] -> byte[], coder
    - Either ISO-8859-1/Latin-1

    - Either UTF-16 if required

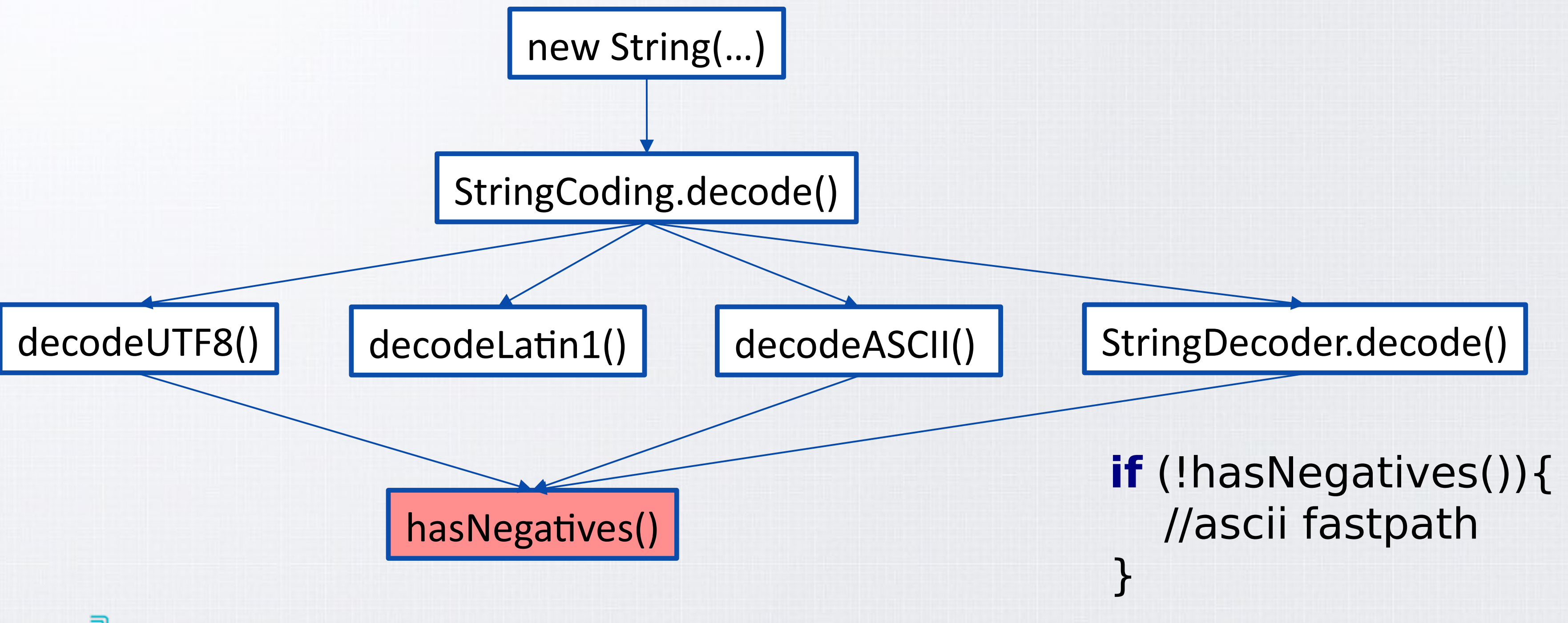
1001 Heap Dump



- Log-normal distribution
- < 0.3% of all strings are not Latin-1
- 18% strings < 8 symbols
- 66% strings < 32 symbols
- 95% strings < 128 symbols

Any changes to improve the current state of things should not case regressions on this dataset

String s = new String(“Can this work faster?”);




```
if (!hasNegatives()){
   //ascii fastpath
}
```

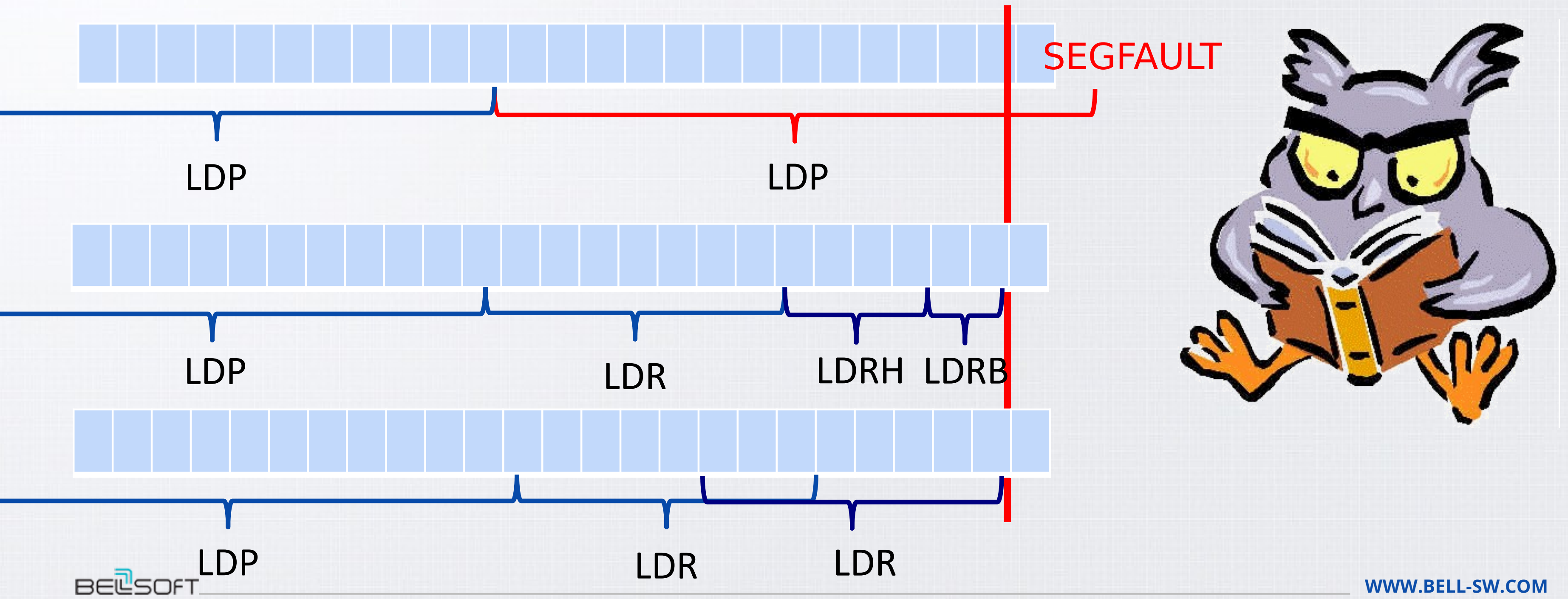
StringCoding.hasNegatives()

```java
@HotSpotIntrinsicCandidate
public static boolean hasNegatives(byte[] ba, int off, int len) {
    for (int i = off; i < off + len; i++) {
        if (ba[i] < 0) {
            return true;
        }
    }
    return false;
}
```

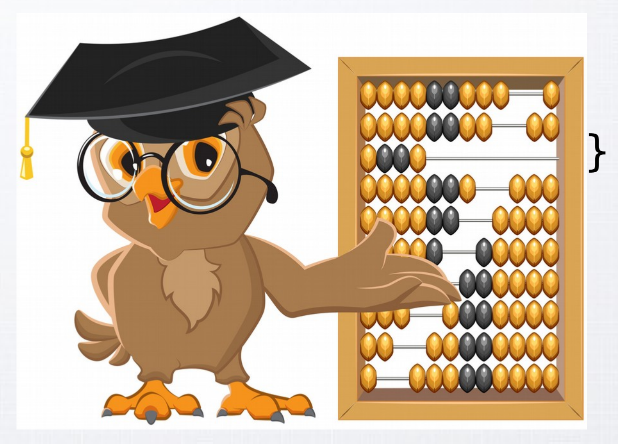
Some ARM assembly – memory reads

	Register	Width (bits)	Latency (cycles)
LDRB	GPR	8	4
LDRH	GPR	16	4
LDR	GPR	32 or 64	4
LDP	GPR	64+64	5

Learning to read (again)



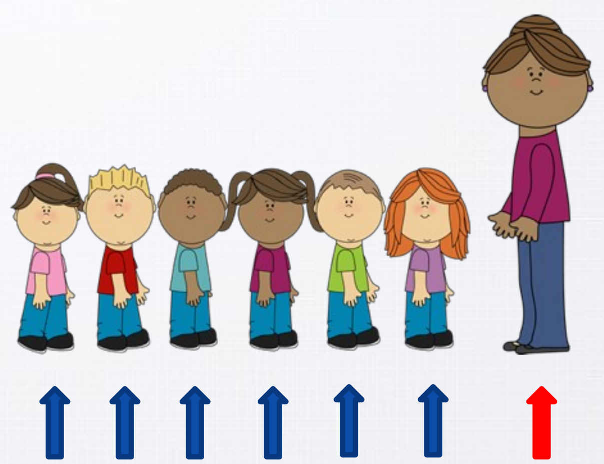
And compare 8 bits at a time with 0



```
for(int i = off; i < off + len; i++) {
    if (ba[i] < 0) {
        return true;
    }
}
```

```
const uint64_t UPPER_BIT_MASK=0x8080808080808080;
...
__ tst(rscratch2, UPPER_BIT_MASK);
```

Aligned memory access



x86:

- in most cases modern processors do not have a penalty for unaligned memory access

ARM is a spec:

- some CPU manufacturers do not have a penalty
- others do have (20%, 50%, 100%)

How to align memory access

```
// pre-loop
    __ ldr();
    ...
    __ tst(..., UPPER_BIT_MASK);
// main loop
    __ ldp(); //aligned
    ...
    __ tst(..., UPPER_BIT_MASK);
```

LDR

LDP

The plan for hasNegatives() intrinsic

- Read as much bytes at a time as possible, without crossing the page boundaries
  - If the page border is close
    - Read less bytes
    - Shift to the left
- Compare as many bytes with 0 as possible at a time
- Align memory access

- Reality
  - The code gets too big – 200 instructions
    - This interferes with inlining: C2 inlines up to1500 instructions

Code is too big – what do we do?

- ARM ASM pseudo-code in Java that is short (27 instructions)
  - Not optimal, unaligned, but short

```
if (len > 32)
    return stubHasNegatives(ba, 0, len);
for (int i = 0; i < 32; i++) {
    if (ba[i] < 0) {            // ldr, tst
        return true;
    }
}
return stubHasNegatives(ba, 32, len); // ldp, tst
```

- The rest of the code goes to stub

What is a stub?

- A type of assembly inline in HotSpot
- Close analogy is a function
  - Can be called from macroAssembler
  - Code gets loaded during JVM startup once
  - Does not get inlined
- Several entry points are possible
- Some performance penalty calling stub

What should we place in stub?

```
// align memory access
__ bind(LARGE_LOOP); // 64 byte at a time
4x  __ ldp(); //ary1, ary1+16, ary1+32, ary1+48
    __ add(ary1, ary1, large_loop_size);
    __ sub(len, len, large_loop_size);
7x  __ orr(...);
    __ tst(tmp2, UPPER_BIT_MASK);
    __ br(Assembler::NE, RET_TRUE);
    __ cmp(len, large_loop_size);
    __ br(Assembler::GE, LARGE_LOOP);
```

OK, we helped C2. Can we help the hardware?

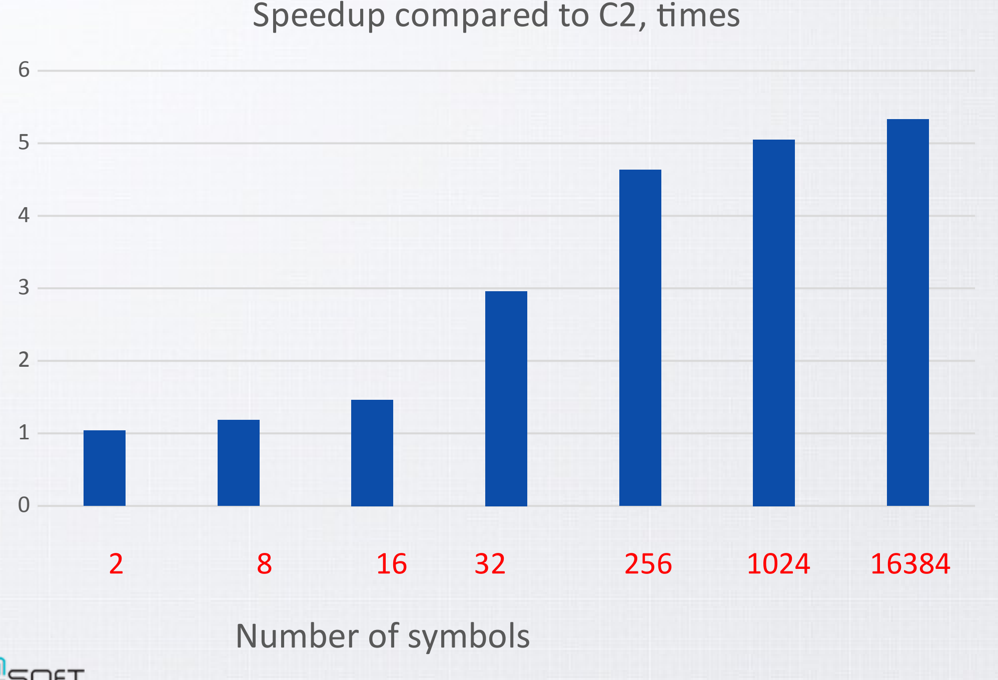
Software Prefetching

Let's give a processor a hint where we are going to read from memory next time:

```
__ prfm(Address(ary1, SoftwarePrefetchHintDistance));
   // do local register or operations on data in cache
__ ldp();
```

- Can be a major performance gain if
  - Processor has enough data to process between prfm and memory load
  - SoftwarePrefetchHintDistance is correctly defined:
    > d_cache_line_size

Benchmark for new String() – long strings



Longer string sizes experience more performance gain from optimization due to

- Optimal ldp & tst use
- Prefetching

Speedup up to 5x!

Benchmark for new String() – results

3
2.8
2.6
2.4
2.2
2
1.8
1.6
1.4
1.2
1

0.06
0.05
0.04
0.03
0.02
0.01
0

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Improvement over C2, times | Length Probability

OpenJDK

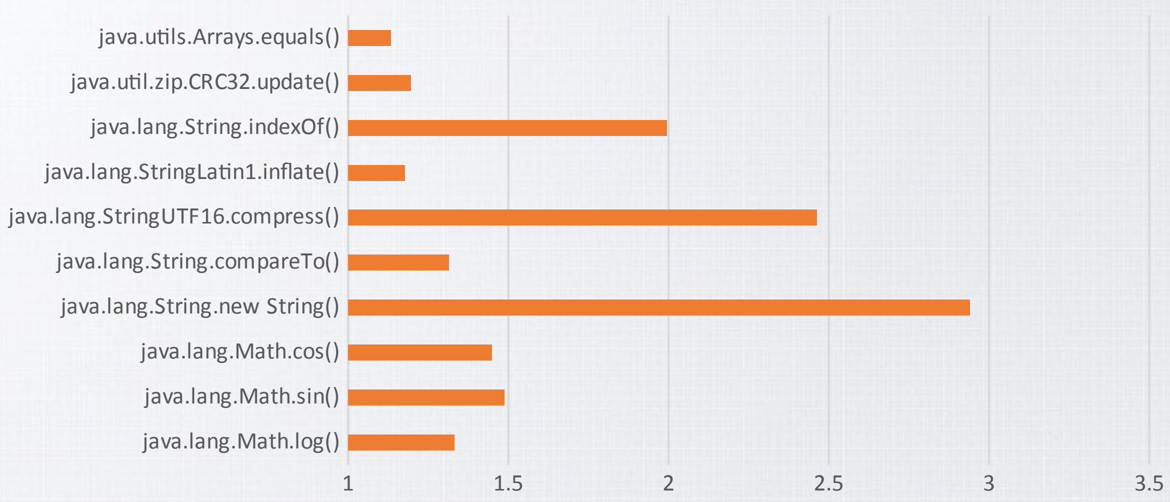
JEP 315: Improve Aarch64 Intrinsics

Owner	Dmitrij Pochepko
Type	Feature
Scope	Implementation
Status	Closed / Delivered
Release	11
Component	hotspot / compiler
Discussion	hotspot dash compiler dash dev at openjdk dot java dot net
Effort	L
Duration	L
Reviewed by	Mikael Vidstedt, Vladimir Kozlov
Endorsed by	Vladimir Kozlov
Created	2017/10/10 12:40
Updated	2018/09/10 14:45
Issue	8189104

Summary

Improve the existing string and array intrinsics, and implement new intrinsics for the `java.lang.Math` sin, cos and log functions, on AArch64 processors.

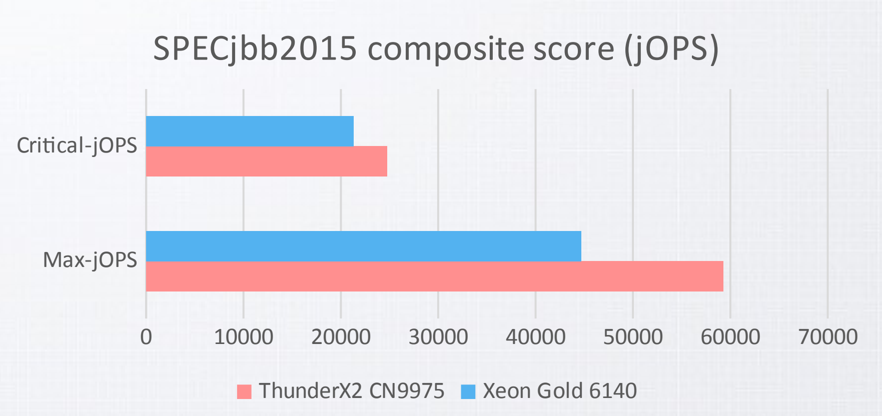
Performance improvement



- Speedup up to 78x in microbenchmarks

* mean improvement over different size, length, encondings

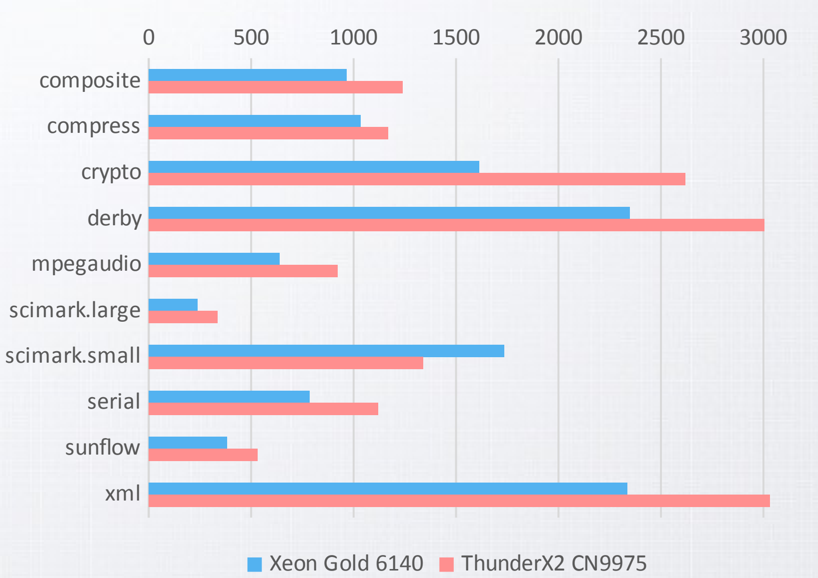
JVM Benchmark #1 results



- Liberica JDK 11
- Average over 20 runs
- JEP 315 in JDK 11
- Cavium Thunder X2 outperforms Xeon 6140
  - by 33% in Max-jOPS score
  - by 16% in Critical-jOPS score

ARMv8: -Xmx24G -Xms24G -Xmn16G -XX:+AlwaysPreTouch -XX:+UseParallelGC -XX:+UseTransparentHugePages -XX:-UseBiasedLocking
X86: -Xmx24G -Xms24G -Xmn16G -XX:+AlwaysPreTouch -XX:+UseParallelGC -XX:+UseTransparentHugePages -XX:+UseBiasedLocking

JVM Benchmark #2 results



- Liberica JDK 11
- Default JVM settings
- Average over 20 runs
- Thunder X2 outperforms Xeon 6140
  - by 62% in Crypto
  - by 42% in MpegAudio
  - By 29% in XML
  - by 12% in Compress
- Xeon 6140 outperforms Thunder X2
  - By 29% in scimark.small

Where to try ARM servers?

Bare Metal

VPS

VPS

Conclusions

- Arm server vendors did a great job
- Cloud providers provide access to Arm servers right now
- Ubuntu, Red Hat, Oracle Linux, SuSE have ARMv8 support
- The software ecosystem just works as expected on ARMv8
- OpenJDK 11 is optimized for ARMv8

Download and install Liberica for ARMv8