Java on ARM
Theory, Applications and Workloads

Dmitry Chuyko
BellSoft JVM Team
September 21, 2018
Whoami

Dmitry Chuyko

@dchuyko  @bellsoftware

Liberica – Java 10 for Raspberry Pi
http://bell-sw.com

Former employer:

ORACLE
Outline

1. We contribute
2. Darling, shall we try ARM?
3. What is Java SE on ARM?
4. Darling, we will have a JEP!
5. Optimization in detail
6. Is nothing sacred?
7. Evaluation
8. Next JDK
We contribute
We contribute: Tricky statistics

External contributions to OpenJDK upstream Aug ’17 - Aug ’18
*Note: Oracle contributed ~3965 patches in the same period

Red Hat | SAP | Google | BellSoft | SUNY Oswego | IBM | NTT | ARM | Qualcomm | Intel | Linaro | Amazon | JetBrains | Longsoon | Azul | Alibaba | SuSE | Twitter

0  | 50  | 100  | 150  | 200  | 50  | 100  | 150  | 200  | 50  | 100  | 150  | 200  | 50  | 100  | 150  | 200  | 50  | 100  | 150  | 200
We contribute: OpenJDK AArch64

Cavium ThunderX
- 2x48x1, 1x48x1
- ARMv8-A
- 2.5 GHz
- Server SoC
Available since 2015

Cavium ThunderX2
- 2x32x4, 1x32x4
- ARMv8.1-A
- 3.0 GHz
- Server SoC
Available since May 2018
We contribute: OpenJDK ARM32

Raspberry Pi
- Raspberry Pi 3
- Raspberry Pi 2
- Raspbian
- Oracle JDK 8

Liberica JDK
- JDK 10 and 9
- JDK 11 pre-release
- Linux ARMv6 hf
- Linux ARMv8, x86_64

Available since 2017
We contribute: Liberica

- Open source, OpenJDK
  github.com/bell-sw/Liberica
- Technology Compatibility Kit (TCK) compliant
  Tested on real hardware
- OpenJFX 10 and Device IO API
  bell-sw.com/liberica-release-notes.html
- JDK and JRE distributions, Jigsaw
- Client, server and minimal VM
Darling, shall we try ARM?
Something new: ARM64 servers

Consumer perspective
- Cheap
- Not x86
- Many cores
- Low power consumption

JDK dev perspective
- Rare
- Diverse
- Different ISA and memory model

Write once, run anywhere
Something new: ARM64 servers

Consumer perspective
• Cheap
• Not x86
• Many cores
• Low power consumption

JDK dev perspective
• Rare
• Diverse
• Different ISA and memory model

Write once, run anywhere
Provide excellent performance
Something new: ARM64 CPUs

We target

1. Cavium ThunderX2
2. Cavium ThunderX
3. Arm Cortex-A family
Something new: ARM64 CPUs
Where to get one

• Provided HW vendor
• Public cloud
  – Packet (bare metal, packet.net)
  – Scaleway (virtual, scaleway.com)
  – More on the way?
• Own lab
What is Java SE on ARM
Java on ARM: Ecosystem

- **OpenJDK**
  - **Hotspot**
    - Ports: arm64, aarch64, arm32, aarch32
    - Distros: Oracle, openjdk-N-xxx, Liberica, Zulu etc.
    - Graal
  - **J9**
- **GraalVM, Excelsior JET etc.**
- **Linux**
Java on ARM: Something we started with in 2017

- Hotspot arm64 port (JDK 8+)
- Hotspot aarch64 port (JDK 9+)

Part of intrinsics analysis
Java on ARM: Something we started with in 2017

- Stable hardware
- Working OS
- (Somewhere) working PMU events
- Almost working toolchains and ports
  - Build issues
  - Missing features
  - Instruction encoding bugs
- A lot of optimization already made

Gratitude
- Oracle
- Red Hat
- Linaro
- Cavium
Java on ARM: Something we see now

- Clusters
  - Isambard, Sandia
- Supercomputers
  - Post-K
- Clouds
- BigData
- Competitive benchmark scores
- Applications
  - Hadoop, Tomcat, Spark, Kafka, Cassandra, Lucene, Flink...
Darling, we will have a JEP!
JEP 315: Description

- JDK-8189104. JEP 315: Improve Aarch64 Intrinsics
  - Bellsoft & Cavium
  - openjdk.java.net/jeps/315
  - Performance is motivation
  - Missing intrinsics
  - Not optimal code
JEP 315: Some results

- Actual speed-up in microbenchmarks is up to 78x
- Positive impact on Hadoop and common Java tasks (XML, data decoding/encoding)

* Geomean gain across different sizes, length, and encoding
JEP 315: Sharpening tools

- Micro-benchmarks
  - JMH
  - Native

- Industry standard benchmarks
- Target workloads
  - Terasort

- Recent workloads on recent JDK
- Profilers
Profiling: Some tools for Hotspot on AArch64

- Linux perf, BPF (newer kernels)
- Java stack sampling
- JDK 9+ (aarch64 port). Async profiler
  - JDK-8201564: Make sure that Flight Recorder works
- JDK 8 (arm port). Honest profiler
  https://github.com/jvm-profiling-tools/honest-profiler
  - Fixed #239: No support for Arm64
  - PR 241
Profiling: Async profiler on ARM64

https://github.com/jvm-profiling-tools/async-profiler

- Low overhead, precise
- Mix of `perf` and `AsyncGetCallTrace`
- Added generic AArch64 support
  - 👇 PR 54, PR 55
- Fixed T99 issues
  - 👇 #101: 64 KB pages support
Profiling: Async profiler on arm64

In general async-profiler supports JDK 8. But in particular on arm port you can get

```c
frame::sender <sp=0xfffffa5b2dce0, %unext_sp=0xfffffa5b2dce0, fp=0xfffffa5b2dce0, %pc=0xffffffff
--- >  <sp=0xfffffa5b2dcf0, fp=0xfffffa5b2dde0, pc=0xfffffa62f2d44>
```

#
# A fatal error has been detected by the Java Runtime %Environment:
#
# Internal Error (frame_arm.cpp:479), pid=20405, %tid=0x0000ffffa5b30200
# Error: Unimplemented()

So use up-to-date Java or different profiler.
Optimization in detail
In detail: One of Java inintrinsics example

CRC32
- @since 1.1
- Backed by zlib
- C implementation
- Good speed
- x86 intrinsic
  - @since 8
- ARM64 intrinsic
  - @since 9

CRC32C
- @since 9
- Java algorithm
- Slicing by 8 with table lookups
- Good speed
- x86 intrinsic
  - @since 9
- ARM64 intrinsic
  - @since 9
In detail: Different CPU capabilities

- JVM picks proper version regarding of CPU and options
- NEON
- CRC32, CRC32C
In detail: Checksums of big data

- Cassandra. SSTable digest
  - Adler32, CRC32
- Kafka. Integrity of every message
  - CRC32, CRC32C
- HBase. Integrity of every HFileBlock
  - CRC32, CRC32C
- Hadoop. Core code, HDFS
  - CRC32, CRC32C
In detail: Checksums of big data

HDFS

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>hdfs-site.xml</td>
<td></td>
</tr>
<tr>
<td>dfs.blocksize</td>
<td>134217728</td>
</tr>
<tr>
<td>dfs.checksum.type</td>
<td>CRC32C</td>
</tr>
<tr>
<td>dfs.bytes-per-checksum</td>
<td>512</td>
</tr>
</tbody>
</table>

- HADOOP-6148 Implement a pure Java CRC32 calculator
- HADOOP-6166 Improve PureJavaCrc32
- HADOOP-7333 Performance improvement in PureJavaCrc32
- HDFS-6903 Crc32 checksum errors in Big-Endian Arch
  ...
In detail: Hadoop-common

DataChecksum

- JDK 8+
- PureJavaCrc32, PureJavaCrc32C
- Knows j.u.z classes
- Bulk CRC check in native library
In detail: Benchmarks

- ARM64
- Linux kernel 4.8, 4.10
- JDK 9 intrinsic as a baseline
- JMH 1.19
  
  http://openjdk.java.net/projects/code-tools/jmh/
Benchmarks: Basic benchmark

@Param
int size;

byte[] bytes;
ByteBuffer buf;
CRC32 crc32;

@Setup
public void setup() {
  bytes = new byte[size];
  new Random(seed).nextBytes(bytes);
  buf = ByteBuffer.wrap(bytes);
  crc32 = new CRC32();
}

@Benchmark
public long calcCRC32() throws NoSuchAlgorithmException {
  crc32.update(bytes);
  return crc32.getValue();
}
Benchmarks: CRC32 speed. Variants

Implementations

- CRC32 instruction is fastest
- Slowdown beyond L1 size
- NEON is much faster than plain arithmetic
Benchmarks: CRC32 speed. Optimized?

Software Prefetching

- PRFM instruction
- 2x speedup on large buffer
- 4% regression on 512 B
  - Extra instruction, branch or mis-prefetch
- T88 only
  - Other μ-archs have hardware prefetching
Benchmarks: Walk buffer, no copying

```java
static final int CHUNK_SIZE = 64*1024*1024;
byte[] fileChunk;
int pos;

@Benchmark
class CalcNextInArray {
    public long calcNextInArray() throws NoSuchAlgorithmException {
        CRC32 crc32 = this.crc32;
        crc32.reset();
        crc32.update(fileChunk, pos, size);
        pos = (pos + size) % chunkLen;
        return crc32.getValue();
    }
}
```
Benchmarks: Walk buffer CRC32 speed. Optimized?

Software Prefetching

- PRFM instruction
- 2.7x speedup on large buffer
- No degradation
- Wrong benchmark!
CRC32C: Intrinsic main loop. JDK9 AArch64

align(CodeEntryAlignment);
BIND(CRC_by64_loop);
subs(len, len, 64);
ldp(tmp, tmp3, Address(post(buf, 16)));
crc32cx(crc, crc, tmp);
crc32cx(crc, crc, tmp3);
ldp(tmp, tmp3, Address(post(buf, 16)));
crc32cx(crc, crc, tmp);
crc32cx(crc, crc, tmp3);
ldp(tmp, tmp3, Address(post(buf, 16)));
crc32cx(crc, crc, tmp);
crc32cx(crc, crc, tmp3);
ldp(tmp, tmp3, Address(post(buf, 16)));
crc32cx(crc, crc, tmp);
crc32cx(crc, crc, tmp3);
br(Assembler::GE, CRC_by64_loop);

- Special instruction
- Loop unrolling
- Code alignment
align(CodeEntryAlignment);
BIND(CRC_by64_loop);
subs(len, len, 64);
crc32cx(crc, crc, tmp2);
1dr(tmp0, Address(buf, 8));
crc32cx(crc, crc, tmp3);
1dr(tmp1, Address(buf, 16));
crc32cx(crc, crc, tmp0);
1dr(tmp2, Address(buf, 24));
crc32cx(crc, crc, tmp1);
1dr(tmp3, Address(buf, 32));
crc32cx(crc, crc, tmp2);
1dr(tmp0, Address(buf, 40));
crc32cx(crc, crc, tmp3);
1dr(tmp1, Address(buf, 48));
crc32cx(crc, crc, tmp0);
1dr(tmp2, Address(buf, 56));
crc32cx(crc, crc, tmp1);
1dr(tmp3, Address(pre(buf, 64)));
br(Assembler::GE, CRC_by64_loop);

- Special instruction
- Loop unrolling
- Code alignment
- Software pipelining
- Data dependency break
- Weaker data alignment requirements
CRC32C: JDK10 over JDK9. AArch64

- JDK-8189176. Improve _updateBytesCRC32 intrinsic
- JDK-8189177. Improve _updateBytesCRC32C intrinsic

- x1.5 speed-up on 512 B block on Thunder X
- x1.2 speed-up on 512 B block on fully utilized Thunder X2
- x1.6 speed-up on 4 KB block on T88
- x2.5 speed-up on 16 KB block on Cortex A53

Released in March 2018
CRC32C: JVM warmup

JDK-8189745. AARCH64: Use CRC32C intrinsic code in interpreter and C1

- x30-50 speed-up for interpreter
- x7 speed-up for C1

Released in March 2018
Is nothing sacred?
Wider contribution: Graal
JDK 11

- PR 268: Implement CRC32C substitutions
  - \(x^3\) speed-up for Graal
  - AArch64, x86, SPARC

- PR 319: String.compareTo for AArch64

Already in JDK mainline
- \(-XX:+UseJVMCICompiler\) already works on ARM64

To be released this fall in JDK 11 LTS
Wider contribution: Hadoop-common

- JDK 9 code is good
- 10 code is even better
- But home grown implementation is used

Hadoop Common / HADOOP-15033
Use java.util.zip.CRC32C for Java 9 and above

Released in v3.1, April 2018
Evaluation
Profiling: Generic workload setup

- Hadoop 3.x
- Without libhadoop.so for the platform
- YARN
- Pseudo-distributed mode
- Updated HiBench 7.1-SNAPSHOT

https://github.com/intel-hadoop/HiBench
Profiling: Hadoop

- OpenJDK 11 AArch64
- Hadoop 3.1-SNAPSHOT
- Terasort
- async-profiler for all YARN JVMs
  - 5 sec dump intervals
  - CPU flame graphs
Profiling: Terasort-large before and after. T88, JDK11

Find 10 differences
Proportion predicts ≈1% overall improvement
Profiling: Terasort-large. T88, JDK11

Actual result is $\approx 5\%$ improvement on average

Released in v3.1, April 2018
Benchmarks: Crc32PerformanceTest
v3.2.0-SNAPSHOT, JDK11

x86-64 1x4x1 AVX2 (256 bit) SSE4.2

<table>
<thead>
<tr>
<th>#T</th>
<th>Java crc32</th>
<th>Java crc32c</th>
<th>Hadoop crc32</th>
<th>Hadoop crc32c</th>
<th>Native crc32</th>
<th>Native crc32c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5860</td>
<td>3648</td>
<td>1853</td>
<td>1571</td>
<td>2000</td>
<td>14001</td>
</tr>
<tr>
<td>2</td>
<td>4976</td>
<td>3160</td>
<td>1627</td>
<td>1430</td>
<td>1853</td>
<td>9542</td>
</tr>
<tr>
<td>4</td>
<td>4475</td>
<td>3209</td>
<td>1682</td>
<td>1463</td>
<td>1929</td>
<td>5265</td>
</tr>
<tr>
<td>8</td>
<td>2554</td>
<td>1835</td>
<td>926</td>
<td>740</td>
<td>959</td>
<td>3455</td>
</tr>
<tr>
<td>16</td>
<td>1084</td>
<td>1159</td>
<td>506</td>
<td>433</td>
<td>558</td>
<td>1983</td>
</tr>
</tbody>
</table>

- NativeC is fastest
  - x3.8 over JavaC
  - Instruction vs AVX2
  - Bulk vs sequential
- CRC32C vs CRC32
- Not as good at scale
- Intrinsics vs Hadoop
  - x2.2

bytes: 512; units: MB/sec
Benchmarks: Crc32PerformanceTest v3.2.0-SNAPSHOT, JDK11

T88 (in-order)

<table>
<thead>
<tr>
<th>#T</th>
<th>Java crc32</th>
<th>Java crc32c</th>
<th>Hadoop crc32</th>
<th>Hadoop crc32c</th>
<th>Native crc32</th>
<th>Native crc32c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>802</td>
<td>1163</td>
<td>310</td>
<td>299</td>
<td>426</td>
<td>1065</td>
</tr>
<tr>
<td>2</td>
<td>830</td>
<td>820</td>
<td>294</td>
<td>280</td>
<td>628</td>
<td>763</td>
</tr>
<tr>
<td>4</td>
<td>1056</td>
<td>1023</td>
<td>292</td>
<td>279</td>
<td>691</td>
<td>611</td>
</tr>
<tr>
<td>8</td>
<td>1028</td>
<td>980</td>
<td>281</td>
<td>273</td>
<td>749</td>
<td>657</td>
</tr>
<tr>
<td>16</td>
<td>934</td>
<td>1063</td>
<td>300</td>
<td>278</td>
<td>636</td>
<td>962</td>
</tr>
</tbody>
</table>

bytes: 512; units: MB/sec

- JavaC is fastest
- Intrinsics vs Hadoop – x4
## Benchmarks: Crc32PerformanceTest
### v3.2.0-SNAPSHOT, JDK11

**T99 (out-of-order)**

<table>
<thead>
<tr>
<th>#T</th>
<th>Java crc32</th>
<th>Java crc32c</th>
<th>Hadoop crc32</th>
<th>Hadoop crc32c</th>
<th>Native crc32</th>
<th>Native crc32c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2871</td>
<td>2885</td>
<td>599</td>
<td>625</td>
<td>4437</td>
<td>4442</td>
</tr>
<tr>
<td>2</td>
<td>2832</td>
<td>2837</td>
<td>596</td>
<td>622</td>
<td>4133</td>
<td>4138</td>
</tr>
<tr>
<td>4</td>
<td>2819</td>
<td>2831</td>
<td>597</td>
<td>622</td>
<td>3953</td>
<td>3938</td>
</tr>
<tr>
<td>8</td>
<td>2807</td>
<td>2828</td>
<td>597</td>
<td>622</td>
<td>3821</td>
<td>3809</td>
</tr>
<tr>
<td>16</td>
<td>2811</td>
<td>2818</td>
<td>597</td>
<td>622</td>
<td>3707</td>
<td>3699</td>
</tr>
</tbody>
</table>

- **NativeC is fastest**
  - x1.5 over JavaC
  - Bulk vs sequential
- **CRC32C vs CRC32**
- **Good scalability**
- **Intrinsics vs Hadoop**
  - x4.6

**bytes: 512; units: MB/sec**
JFR & JMC: Node manager processes
JFR & JMC: Node manager threads
JFR & JMC: Data node threads
JFR & JMC: Map profile
JFR & JMC: External view of JVM processes timeline
Next JDK
Next JDK: JDK 10 & 11 ported performance features
Besides just intrinsics

- Compact strings
- AOT (ahead of time compilation)
- AppCDS
  - Bellsoft
Next JDK: JDK 11 evaluation

SPECjbb2015-Composite performance results on single-socket Xeon Gold 6140 and ThunderX2 CN9975 with DDR4-2666 memory running Ubuntu 16.04. Higher is better.
Next JDK: JDK 11 evaluation

SPECjvm2008 performance results on single-socket Xeon Gold 6140 and ThunderX2 CN9975 with DDR4-2666 memory running Ubuntu 16.04. Higher is better.
Next JDK: JDK 12

- ZGC
- Panama
  - SVE
- Valhalla
  - Bellsoft already experiments with Value Types