Java on Arm processors: Understanding AArch64 vs. x86

Arm-based processors are increasingly popular and are in the news thanks to Apple’s latest notebooks and Oracle’s cloud services.

by Aleksei Voitylov

January 15, 2021

[The Arm processor architecture is in the news because Apple’s newest notebooks and desktop computers, introduced in late 2020, use Apple’s own Arm-based M1 system-on-a-chip platform. Due to popular interest, Java Magazine is reprinting an article first published in the September/October 2018 issue about Java on Arm, which of course predates Apple’s M1. Oracle has also announced plans to bring 160-core Arm servers to its cloud. We have made only a few updates to the original piece; we plan to present another article about Java on Arm-based processors later in 2021. —Ed.]

For many years, Arm-based processors were mainly viewed as targeting the embedded market, because they offer sufficient performance while keeping power consumption low. But many hardware vendors are now using the 64-bit Arm architecture, called AArch64, to build server CPUs and to compete with the x86 architecture in the cloud and in high-performance computing.

This range of deployment platforms adds to the complexity of the Java Arm port, because the port must support a variety of CPU vendors and workloads.

In this article, I explore the evolution of Java and the Java ecosystem and their status on Arm architectures. I also discuss some recent developments in Java features and performance for Arm processors, emphasizing both server and IoT/embedded deployments.
The state of the Arm architecture

Leaving aside the embedded and mobile markets, where Arm dominates with its 32-bit instruction set architectures (ISAs), it’s no longer stretching the point to say that Arm provides a viable alternative for markets that are currently dominated by the x86 architecture.

Unlike microprocessor vendors such as Intel or AMD that focus on shipping processors, Arm is primarily an architecture design company selling architectural and core licenses to its customers, which turn that intellectual property into actual silicon. This model allows a great variety of actual implementations of the same architecture to coexist and compete in different market segments.

It is clear from recent developments in the Arm architecture itself that the focus has shifted to competitive Arm-based server CPU designs. In 2016, Arm finalized a 64-bit- and 32-bit-capable ARMv8-A ISA that targets both the embedded and server markets. This architecture mandated the presence of a single instruction, multiple data (SIMD) instruction set (called NEON) and introduced optional instructions for AES encryption and for SHA-1, SHA-256, and CRC32 calculations, which some vendors use to boost cryptographic and checksum performance.

In 2017, Arm extended this architecture by adding new atomic instructions. Later, the ARMv8.2-A ISA added half-precision floating-point data processing and specialized SIMD instructions that improve the performance of machine learning computations. In addition, in the ARMv8.2-A ISA, optional Scalable Vector Extension (SVE) instructions were introduced for better support of vectorization (compared to the NEON instruction set), thereby making the ARMv8 architecture much more suitable for technical computing. Most recently, the ARMv8.3-A ISA added SIMD complex-number support.

The ARMv8 architecture leaves room for vendor design selection to achieve performance, complexity, and power goals. It adopts a relaxed hardware memory model that is weaker than that of x86 processors (which use x86-TSO). Thus, you can observe more out-of-order effects. This architecture also adds new concurrency primitives, including the load-acquire and store-release instructions, as well as weaker barrier instructions. But most Java developers will not notice these changes because the JVM hides them inside the implementation.

Several hardware vendors compete with Intel in the server market with their ARMv8-based processor design. Some hardware vendors are already established in the Arm-based server market and have delivered 64-bit production systems used in data centers for several years now.

In technical computing, Sandia National Labs deployed an Arm-based supercomputer called Astra, which has a theoretical peak
of more than 2.3 petaflops. All major Linux distributions support Arm. All the tooling at the operating system and kernel level is stable and ready for production use.

**Availability of Java on Arm architectures**

You will find a good choice of providers for Java and OpenJDK binaries for Arm-based architectures. The Java ports for both the ARMv7 and ARMv8 ISAs are fully functional, and the codebases are available from OpenJDK under the GPLv2.1 license with the classpath extension, which enables most Linux distributions to bundle them.

If your favorite Linux distribution does not contain the required packages or you are looking for commercial support, an excellent set of Java/OpenJDK binaries is provided by several organizations, including BellSoft and Oracle.

**Features of the Java ports**

Although it is very important to ensure the compatibility of Java implementations, passing the Java Compatibility Kit test suite is not the only requirement for a successful Java port.

To meet startup and throughput performance expectations, Java ports for both the ARMv7 and ARMv8 ISAs implement C1 and C2 just-in-time (JIT) compilers, thus allowing them to produce optimized code that takes advantage of the underlying architecture specifics.

On top of that, the `-XX:+TieredCompilation` command-line option is supported and turned on in the server VM, which allows faster startup and higher C2 throughput. A full set of garbage collectors is supported in both the ARMv7 and ARMv8 Java ports.

For embedded use cases, some ARMv7 ports include a lightweight minimal VM. On JDK 9 or higher, the new Java modules enable building Java runtime images that have a small static footprint. Running the following commands on the BellSoft ARM JDK 10, for example, produces a Java runtime with the `java.base` module that has a static footprint as small as 16 MB. Surprisingly, `java.base` (perhaps with the addition of several other modules) is sufficient for many Java applications that are tailored for constrained IoT gateways. For example, a runtime capable of running Apache Felix or Jetty fits into 32 MB memory and can be created with the following commands:

```
OUTPUT=~/out
bin/jlink --module-path jmods --compress=2 --
rm -r $OUTPUT/lib/client $OUTPUT/lib/server
echo "-minimal KNOWN" > $OUTPUT/lib/jvm.cfg
```
Over the years, the ARMv8 port received built-in optimized assembly intrinsics for CPU-intensive operations. Those intrinsics were improved by JEP 315 in Java 11.

All the common features that appear in other Java ports also work on Arm, including Docker support and Application Class-Data Sharing (AppCDS) v2, as specified in JEP 310.

Table 1 provides a detailed comparison of major JVM features on x86/64, ARMv8 64-bit, and Arm 32-bit ports.

**Performance of the Arm 64-bit JVM port**

Let’s dive into the performance of the ARMv8 port, because the server market is where performance matters most. To make a valid comparison, it is important to find x86- and Arm-based server equivalents. Luckily, the Cavium ThunderX2 ARMv8 CPU line provides a processor that’s comparable to the Intel Xeon processors based on SPECint2017 rates.

For this comparison, I selected the Cavium ThunderX2 CN9975 and the Intel Xeon Gold 6140 single-socket systems, both equipped with DDR4-2666 memory and running Ubuntu 16.04. (Dual-socket systems with these CPUs are also available.) The ThunderX2 CN9975 CPU has 112 threads (28-core system with 4-way symmetric multiprocessing), and the comparable Intel Xeon Gold 6140 CPU has 36 threads (18-core system with Intel Hyper-Threading).

To assess the performance of the JVM ARMv8 and x86 ports, I ran the widely used SPECjbb2015 1.01 and SPECjvm2008 1.01 benchmarks with OpenJDK 11 EA build 18. All benchmarks were executed 20 times, and the mean values were collected. The SPECjbb2015 benchmark was used to obtain an overall score, while the SPECjvm2008 benchmark provided additional insights into the performance of the ARMv8 64-bit JVM port.

<table>
<thead>
<tr>
<th>Feature</th>
<th>x86/64</th>
<th>ARMv8 (64-bit)</th>
<th>Arm (32-bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMS</td>
<td>CL(env)</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td>JVM</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>VMMA</td>
<td>YES (32-bit)</td>
<td>NO</td>
</tr>
<tr>
<td>JIT</td>
<td>J Compiler</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>GC Compiler</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>Tiered Compilation</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>SMA, Jvm Compiler (Experimental)</td>
<td>YES, since Jan 10</td>
<td>YES, since Jan 11</td>
</tr>
<tr>
<td>GC</td>
<td>SERIAL GC</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>PARALLEL GC</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>CMS GC</td>
<td>YES</td>
<td>DEPRECATED</td>
</tr>
<tr>
<td></td>
<td>Safepoint GC</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>Garbage Collector (GC)</td>
<td>EXPERIMENTAL</td>
<td>IN DEVELOPMENT</td>
</tr>
<tr>
<td>RUNTIME</td>
<td>Suspension Support</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>Uncommit Support</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>Unpause Support</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>Unpause Android Support</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>Serviceability</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>

Table 1. Comparison of features for major x86 and Arm JVM ports

Because the intent of this article is not to report the best score obtainable on a specific hardware system but instead to study
the performance a typical user would see, I intentionally did not fine-tune low-level JVM parameters or kernel settings on either system. Check the SPEC scores for the processors, as reported by the hardware vendors, to compare the highest achievable numbers available with JVM options tuning.

**SPECjbb2015 results.** Figure 1 presents the SPECjbb2015 1.01-Composite results (Critical-jOPS and Max-jOPS) for a single-socket Intel Xeon Gold 6140 system and a ThunderX2 CN9975 single-socket system, both with DDR4-2666 memory and running Ubuntu 16.04. (Higher scores are better.)

![SPECjbb2015 score (jOPS)](image)

**Figure 1.** SPECjbb2015-Composite performance results

The JVM command-line options used for these runs are very common for SPECjbb2015 runs. On the Arm-based system, I used the following:

```
```

On the x86-based system, I used this:

```
```

(Switching biased locking off for the ARMv8 architecture and leaving it on for the x86 architecture gave both platforms slightly better results.)

As you can see in **Figure 1**, the OpenJDK 11 ARMv8 port running on the ThunderX2 CN9975 system outperformed the x86 port running on the Intel Xeon Gold 6140 system by 33% for the Max-jOPS score and by 16% for the Critical-jOPS score. This suggests the ThunderX2 system with the ARMv8 JVM port is very suitable for enterprise workloads represented by the SPECjbb2015 benchmark.

To assess per-thread performance, I also limited the number of CPU threads on the ThunderX2 system to be the same as on the Intel Xeon Gold 6140 system, which used only 32% of its CPU threads. Unsurprisingly, in this case the SPECjbb2015 results clearly favored the Xeon Gold 6140 system, giving it a 30% advantage.
**SPECjvm2008 results.** Figure 2 presents the SPECjvm2008 base results for individual benchmarks together with the composite base results for a single-socket Xeon Gold 6140 system and a single-socket ThunderX2 CN9975 system, both of which had DDR4-2666 memory and were running Ubuntu 16.04. (Higher scores are better.) Because the SPECjvm2008 "compiler" subbenchmark has not worked in this suite since JDK 8, the composite geometric mean base score was manually calculated without a "compiler" benchmark result.

As you can see in Figure 2, the OpenJDK 11 ARMv8 port running on the ThunderX2 CN9975 system outperformed the x86 port running on the Xeon Gold 6140 system by 28% in the SPECjvm2008 benchmark composite base score. There are two main reasons for the overall better score on the ARMv8-based system. The first is that the system has a higher memory bandwidth (eight channels compared with six channels on the Xeon Gold 6140 system). The second is related to the work done in the ARMv8 JVM port that allows the full utilization of the CPU potential and extensions.

To gain additional insights, let's explore the scores for individual SPECjvm2008 workloads.

In eight out of nine SPECjvm2008 benchmarks, the ARMv8 results outperformed the Intel processor, and in the remaining result, the Intel processor was faster. The crypto benchmark results clearly favor an ARMv8-based system, giving it a 62% advantage, which could not be attained if the ARMv8 port didn’t fully utilize the AES and SHA extensions available on the Arm chip.

The compress benchmark (in which the ARMv8 system leads by 12%) uses the CRC32C intrinsic. The XML benchmark (in which the ARMv8 processor leads by 29%) and the mpegAudio benchmark (in which the ARMv8 leads by 44%) use the `java.lang.String` and `java.lang.Arrays` intrinsics. Some of these intrinsics were recently improved in JDK 10 and JDK 11 for the ARMv8 system.

It is also important to understand the results for the benchmark where the x86 OpenJDK port did better (by 29%): scimark.small. The reason for that is the benchmark code: The FFT, LU, SOR, and SPARSE scimark subbenchmarks all contain heavy loops and matrix computation code. Over the years, Intel has invested a lot of effort into loop unrolling and vectorization, which allowed the mapping of such code sequences to AVX instructions on x86 processors. This work has not yet been completed for the ARMv8 C2 port, and the absence of a good equivalent to the Intel AVX 512-bit instruction set does not help.
There is definitely some work ahead to bring the ARMv8 port’s scientific workload performance up to par with that of the x86 implementation. However, for regular server-side Java business application workloads (data processing, XML, crypto operations, and so forth), the OpenJDK 11 ARMv8 port running on Cavium ThunderX2 units currently provides better performance compared with the x86 equivalent.

**Performance diagnostics**

Performance diagnostics tools are essential for understanding the bottlenecks of a Java application being developed or used in production.

Regular performance diagnostics via JDK tools such as Java Management Extensions (JMX) and the JVMTI API work on Arm-based systems just as they do on x86 systems. For a more-thorough Java performance analysis, a group I work with ported the Async Profiler and Honest Profiler to the ARMv8 and contributed the changes back to the project. These ports enabled enhancement of the performance of an application as complex as Hadoop on ARMv8 systems.

If you intend to work on a complex Java application and would like to profile the JVM bottlenecks on the Arm architecture (or any other architecture), these are the open source tools I would recommend.

Java Flight Recorder, which was open-sourced by Oracle and contributed to OpenJDK 11, is also available in Arm-based ports.

**The Java ecosystem on Arm systems**

In theory, all software written in Java should work on all Arm-based systems. However, some big projects make specific tweaks, such as using natively built libraries, that tie them to a specific architecture. The following popular projects, although not claiming official support for the ARMv8 ISA, were tested and work on Arm systems without modification: Hadoop 3.1.0,
Tomcat 9.0.8, Spark 2.3.0, Kafka 1.1.0, Cassandra 3.11.2, Lucene 7.3.0, and Flink 1.4.2.

Several companies—including Arm itself, Azul, BellSoft, Cavium, Linaro, Oracle, Red Hat, and others—collaborate in the OpenJDK codebase to ensure the long-term future of Arm-based ports. This work includes gradual improvements in performance and stability, as well as work on a fully supported GraalVM and Graal as a JIT compiler on ARMv8 processors. Future projects such as Valhalla and Panama will be part of this effort as well.

**Conclusion**

The upstream Arm 32-bit and ARMv8 Java ports are ready for production use, and all of the relevant features are on par with those of x86 platforms.

The 32-bit Arm port provides all the necessary functionality for embedded and IoT deployments, including the C1 compiler for fast startup, a low dynamic memory footprint, and a minimal VM, which allows for the production of Java runtime images that have a small static footprint (under 16 MB). This port works well on such popular devices as the Raspberry Pi and, after proper device and application-specific tuning, the 32-bit Arm port can be used in production under the GPL license.

The ARMv8 port that is aimed primarily at the server market shows better performance results when compared with x86 counterparts on equivalent hardware (a 16% advantage in the SPECjbb2015 Critical-jOPS benchmark, a 33% advantage in the SPECjbb2015 Max-jOPS benchmark, and a 28% advantage in the SPECjvm2008 base composite benchmark). As demonstrated by the SPECjvm2008 benchmarks for typical server-side Java business applications that process and encrypt data and XML files, the OpenJDK 11 ARMv8 port running on a Cavium ThunderX2 system is faster than the Intel counterpart.

The Java software ecosystem is ready for production deployments on Arm-based systems. For embedded and IoT use cases, the Arm platform is already the primary platform of choice, but why would server manufacturers and major cloud providers consider moving to a different architecture if the performance advantage is only tens of percents? Price/performance is the answer.

Given the performance of the JVM on Arm-based systems and the price of the CPUs, this starts to make sense. And it becomes very easy to try—considering how little effort is required to take existing Java applications to a new architecture.

**Dig deeper**

- Update on 64-bit ARM support for Oracle OpenJDK and Oracle JDK
- Java Flight Recorder and JFR Event Streaming in Java 14
Aleksei Voitylov
Aleksei Voitylov is the CTO of BellSoft, where he drives innovation for the company's customers, including optimizing OpenJDK for Cavium systems. Prior to cofounding BellSoft, he ran teams at Oracle and Sun Microsystems and, in particular, he helped deliver multiple components of the JDK 8 and 9 releases, including the HotSpot JVM, and the Java language. He holds a PhD from Saint Petersburg State University.