Accelerating I/O in Cloud – A Data Driven Approach and Case Studies

Yingqi (Lucy) Lu Yingqi.Lu@intel.com

Intel Corporation

Legal notices and disclaimers

- Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer.
- No computer system can be absolutely secure.
- Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit http://www.intel.com/performance.
-
-

Why are we here?

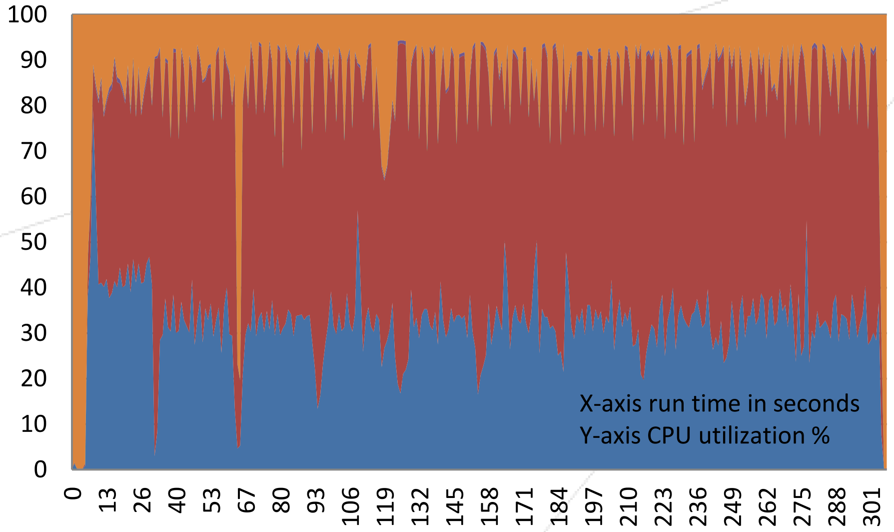
- Modern hardware being continuously developed and adopted into cloud
  - Core count growth
  - Spinning disks to NVMe drives
  - Networking standards evolving faster 10G → 25G → 100G w/ RDMA
- Requires software tuning/optimizations to take full advantage of the hardware is challenging

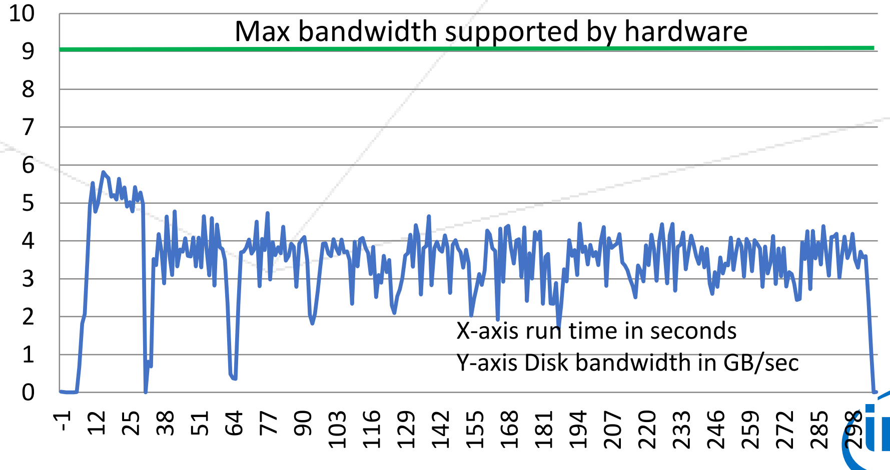
Why are we here?

- Many cloud frameworks are built in Java
- Java I/O is lacking native features as available in C/C++
  - Catching up with new feature enablement in line with modern hardware development
  - New 6 month Java release cadence might help
- Developers
  - Exploring new technologies for performance vs. stay compatible

Apache Cassandra-Stress read performance

- CPU and storage utilization on a tuned performance node (56C, 192GB DRAM, 4 NVMe drives)
- 55% CPU cycles spent in kernel
  - 47% in memory management and IRQ locks
  - Highest function on the call chain: try_to_unmap_one (9.5%) hints to kernel memory page swapping
- Disk 50% utilized: bandwidth and iops





What is being swapped?

- Java uses buffered I/O by default
- All I/O buffered by kernel in DRAM (filesystem cache)
- Kernel constantly refill/cleanup the filesystem cache, especially at high throughput level provided by multi-cores and NVMe drives

Bypass the filesystem cache

- "Direct I/O is a system-wide feature that supports direct reads/writes from/to storage device to/from user memory space bypassing system page cache." – Facebook RocksDB Wiki[1]
- Enabled on many database applications built in C/C++
- Direct I/O support added to Java* SE Development Kit 10
  - GA release on March 2018
  - APIs are designed for easy use and minimal changes to applications

1. https://github.com/facebook/rocksdb/wiki/Direct-IO

Direct I/O's Pros

- No CPU cycles or memory bandwidth spent in copies between filesystem cache and user space
- Avoid filesystem cache thrashing
- Provide consistent I/O throughput and latency
- Avoid redundant caching when application already has its own caching

Direct I/O's Cons

- Direct I/O is not intentioned for traditional spinning devices
- Might not be suitable for sequential I/O which greatly benefits from filesystem cache
- Need extra programming effort to handle the alignment between I/O size, user buffer and storage device block size.

DIRECT I/O Java API

Enum: ExtendedOpenOption

Enum Constant: DIRECT

Description: Flag for Direct I/O defined as one of the ExtendedOpenOption. The flag could be used in FileChannel.open()

Class: FileStore and inherited classes

Method: public int getBlockSize() throws IOException

Description: Return the block size for the disk in bytes. The value could be used for Direct I/O alignment.

Java Code Example – Buffered IO

```
import java.nio.file.Paths;
import java.nio.file.Path;
import java.nio.channels.FileChannel;
import java.nio.ByteBuffer;
import java.nio.file.FileStore;
import java.nio.file.Files;

public class testDirectIO {
  public static void main (String[] args) throws IOException {
    int fileSize = 8192;
    File datafile = File.createTempFile(“myfile", null);
    datafile.deleteOnExit();

    FileOutputStream fos = new FileOutputStream(datafile);
    fos.write(new byte[fileSize]);
    fos.close();
    String path = datafile.getAbsolutePath();
    Path p = Paths.get(path);

    FileChannel newChannel = FileChannel.open(p);

    ByteBuffer buf = ByteBuffer.allocateDirect(fileSize);
    int result = newChannel.read(buf);

    newChannel.close();
  }
}
```

Java Code Example – DIRECT I/O

```
import java.nio.file.Paths;
import java.nio.file.Path;
import java.nio.channels.FileChannel;
import java.nio.ByteBuffer;
import com.sun.nio.file.ExtendedOpenOption;
import java.nio.file.FileStore;
import java.nio.file.Files;

public class testDirectIO {
  public static void main (String[] args) throws IOException {
    int fileSize = 8192;
    File datafile = File.createTempFile(“myfile", null);
    datafile.deleteOnExit();

    FileOutputStream fos = new FileOutputStream(datafile);
    fos.write(new byte[fileSize]);
    fos.close();
    String path = datafile.getAbsolutePath();
    Path p = Paths.get(path);

    FileChannel newChannel = FileChannel.open(p,
ExtendedOpenOption.DIRECT);

    FileStore store = Files.getFileStore(p);
    int alignment = store.getBlockSize();
    ByteBuffer buf = ByteBuffer.allocateDirect(fileSize +
alignment).alignedSlice(alignment);
    int result = newChannel.read(buf);

    newChannel.close();
  }
}
```

Improvements with Direct I/O

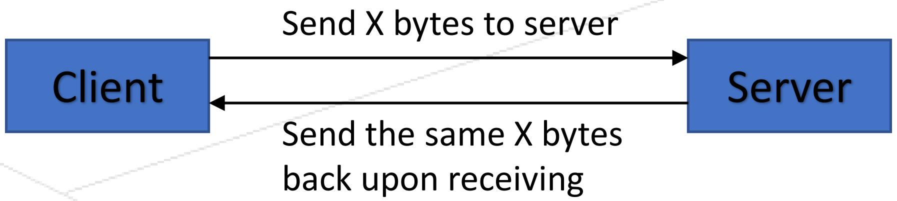
- Kernel time reduce from 55% to 5% → less overhead
- User time increase from 35% to 65% → more meaningful work are done
- Disk bandwidth improved by 2.1x and all 4 NVMe SSDs are fully utilized
- 2.2x throughput improvements on throughput with 90% reduction on 99th percentile latency
- Details on Apache* Cassandra* code changes are available at https://issues.apache.org/jira/browse/CASSANDRA-14466

Who else may benefit from Direct I/O?

- Applications that read randomly
  - A “proof of concept” implemented to Apache HBase* bucket cache
  - Random reads shows up to 2.2x improvement on throughput and 56% reduction on average latency across different load levels
- Applications with build-in cache(s)
  - Ex: Apache Cassandra*, Apache HBase*
- Applications that generate single-use temporarily files
  - Ex: Apache Spark* shuffle service
- Multi-tenanted applications running on the same platform

Network transfer performance

- Micro workload for measuring network latency across different transfer sizes



  - Single threaded
  - Latency is measured at the client side as round trip time
- 35% CPU utilization observed with 32KBytes transfer size on 10Gb NIC
  - 30% are spent in kernel. Mostly handling memory copies and tcp transmissions
- Network device is far from being utilized

TCP/IP networking

- Java supports socket-based networking
  - Based on traditional TCP/IP stack
  - Leverage kernel socket APIs, EX: bind, listen, connect, accept, send and receive
- High kernel utilization is due to multiple back-forth memory copies between kernel and user spaces
- Network bandwidth not scaling with increased device capabilities
- Modern devices need an optimized networking stack for high bandwidth and low latency

Remote Direct Memory Access (RDMA)

- Enable RDMA capable network adapters to transfer data directly to/from application memory
- Data transfers bypass OS kernel
- Avoid multiple data copies between user and kernel spaces
- Permit high-throughput, low-latency networking
- Useful in massively parallel computer clusters

Information Source: https://en.wikipedia.org/wiki/Remote_direct_memory_access

Enable RDMA in Java

- Work-in-progress
  - Java Enhancement Proposal (JEP): http://openjdk.java.net/jeps/337
  - Java Bug System: https://bugs.openjdk.java.net/browse/JDK-8195160
  - Patch under review: http://cr.openjdk.java.net/~ylu/8195160.09/
- Applications aiming at high network throughput and/or low latency may benefit from the feature:
  - Apache* Spark*: shuffle service
  - Apache* HBase* and Apache* Cassandra*: data replication, node repair, peer-peer communication
  - Others

Proposed Java API for RDMA

Class: jdk.net.Sockets

Methods:

- openRdmaSocket: return a RDMA Socket
- openRdmaServerSocket: return a RDMA Server Socket
- openRdmaSocketChannel: return a RDMA SocketChannel
- openRdmaServerSocketChannel: return a RDMA ServerSocketChannel
- openRdmaSelector: return a RDMA channel selector

Java Server Side Code Example with TCP/IP

```
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.nio.ByteBuffer;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.InetAddress;

public class WebServer {
    public static void main (String [] args)
            throws IOException {

        ServerSocketChannel ssc = ServerSocketChannel.open();
        InetAddress addr = InetAddress.getLocalHost();

        InetSocketAddress hostAddress = new InetSocketAddress(addr, 9000);
        ssc.bind(hostAddress);
        SocketChannel client = ssc.accept();
        int xfSize = Integer.parseInt(args[0]);
        ByteBuffer buffer = ByteBuffer.allocate(xfSize);
        int readCount = 0;
        int writeCount = 0;
        int readB = 0;
        int writeB = 0;

        while (readCount < xfSize) {
            readB = client.read(buffer);
            readCount = readCount + readB;
        }
        buffer.flip();
        while (writeCount < xfSize) {
            writeB = client.write(buffer);
            writeCount = writeCount + writeB;
        }
        client.close();
        ssc.close();
    }
}
```

Java Server Side Code Example with RDMA

```
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.nio.ByteBuffer;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.InetAddress;
import jdk.net.Sockets;

public class WebServer {
    public static void main (String [] args)
          throws IOException {

        ServerSocketChannel ssc = Sockets.openRdmaServerSocketChannel();
        InetAddress addr = InetAddress.getLocalHost();

        InetSocketAddress hostAddress = new InetSocketAddress(addr, 9000);
        ssc.bind(hostAddress);
        SocketChannel client = ssc.accept();
        int xfSize = Integer.parseInt(args[0]);
        ByteBuffer buffer = ByteBuffer.allocate(xfSize);
        int readCount = 0;
        int writeCount = 0;
        int readB = 0;
        int writeB = 0;

        while (readCount < xfSize) {
            readB = client.read(buffer);
            readCount = readCount + readB;
        }
        buffer.flip();
        while (writeCount < xfSize) {
            writeB = client.write(buffer);
            writeCount = writeCount + writeB;
        }
        client.close();
        ssc.close();
    }
}
```

Java Client Side Code Example with TCP/IP

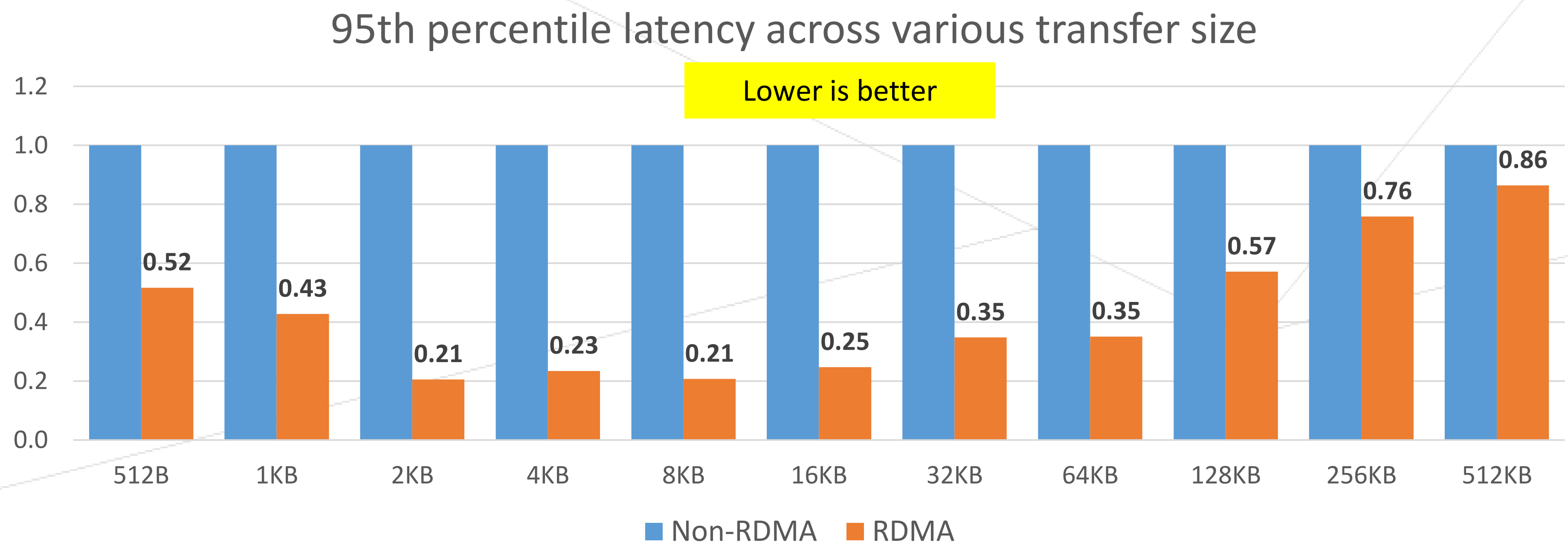
```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;

public class WebClient {
  public static void main(String args[]) throws IOException {
    int xfSize = Integer.parseInt(args[0]);
    InetSocketAddress hostAddress = new
InetSocketAddress("30.30.30.1", 9000);

    SocketChannel client = SocketChannel.open();
    client.connect(hostAddress);

    ByteBuffer buf = ByteBuffer.allocate(xfSize);
    for (int i = 0; i < xfSize; i++) {
      buf.put((byte)'a');
    }
    buf.flip();
    int writeB = 0;
    int writeCount = 0;
    int readB = 0;
    int readCount = 0;

    while (writeCount < xfSize) {
      writeB = client.write(buf);
      writeCount = writeCount + writeB;
    }
    buf.flip();
    while (readCount < xfSize) {
      readB = client.read(buf);
      readCount = readCount + readB;
    }
    client.close();
  }
}
```

Java Client Side Code Example with RDMA

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;
import jdk.net.Sockets;

public class WebClient {
   public static void main(String args[]) throws IOException {
      int xfSize = Integer.parseInt(args[0]);
      InetSocketAddress hostAddress = new
InetSocketAddress("30.30.30.1", 9000);

      SocketChannel client = Sockets.openRdmaSocketChannel();
      client.connect(hostAddress);

      ByteBuffer buf = ByteBuffer.allocate(xfSize);
      for (int i = 0; i < xfSize; i++) {
         buf.put((byte)'a');
      }
      buf.flip();
      int writeB = 0;
      int writeCount = 0;
      int readB = 0;
      int readCount = 0;

      while (writeCount < xfSize) {
         writeB = client.write(buf);
         writeCount = writeCount + writeB;
      }
      buf.flip();
      while (readCount < xfSize) {
         readB = client.read(buf);
         readCount = readCount + readB;
      }
      client.close();
   }
}
```

Improvement with RDMA

- With 32KB transfer size
  - Overall CPU utilization improved from 35% to 60%
  - User space utilization improves from 6% to 47%
  - Memory copies between user and kernel spaces are avoid which contributes to kernel utilization reductions
- Up to 75% reduction on 95th percentile latency



Summary

- I/O infrastructure is key to cloud ecosystem
- New Java libraries and APIs are being developed to scale modern storage and networking hardware devices
- Exploring new features and optimize applications to take full advantage of the hardware

Q/A

Thank you