Accessing InfiniBand hardware from Java using the Foreign Function & Memory API

Filip Krakowski, Fabian Ruhland, Michael Schöttner

Department of Computer Science
Heinrich Heine University Düsseldorf

18/04/2023

Icons made by Smashicons from www.flaticon.com

Motivation

Many data processing and computation frameworks are written in the Java programming language or run on the Java Virtual Machine.

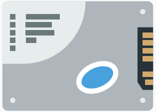
Spark™ **Flink** **Storm**

Samza **Mahout**

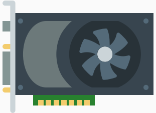
Using traditional sockets these frameworks produce computational overhead regarding the transport layer due to **context switches** / **system calls** and **buffer copies**.

InfiniBand eliminates these problems by allowing applications to **bypass the kernel** and read/write remote memory directly in a zero-copy fashion.

Remote direct memory access (**RDMA**) is not limited to Random Access Memory

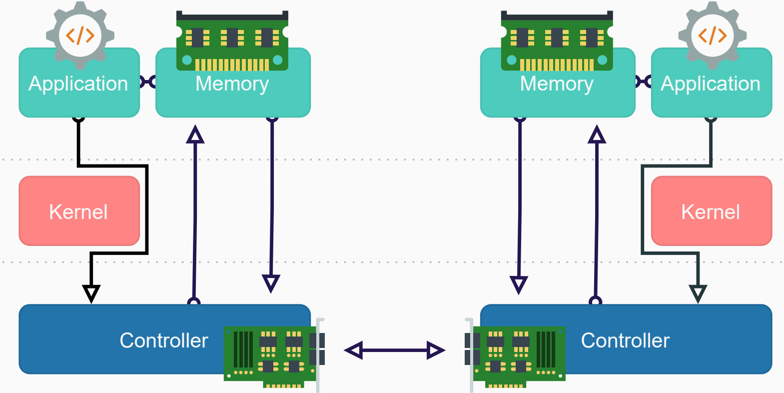


NVMe



GPU

Unlike Ethernet, the InfiniBand protocol stack is implemented within hardware



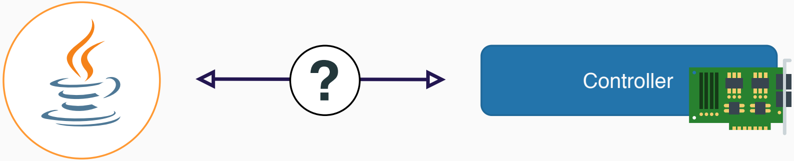
System calls are not necessary, so the kernel can be bypassed

Kernel Bypass

Low latencies
(< 0.7 µs)

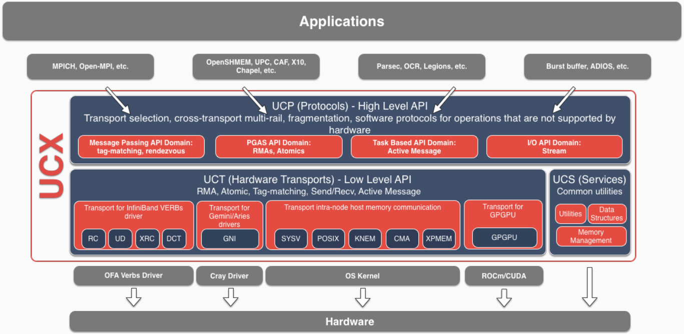
High troughput
(400 Gb/s)

Current JDK implementations do not provide support for InfiniBand hardware



A library is required for interfacing with InfiniBand hardware

The *Unified Communication X (UCX)* [1] Library offers **simple access** to InfiniBand hardware, provides **easy-to-use abstractions** and is **written in C**.



[1] https://github.com/openucx/ucx

Native Interface

Two options for interfacing with native code from Java

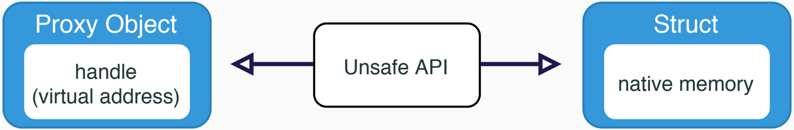
Java Native Interface [2]
(since JDK 1.1 - 1997)

Project Panama [3]
(since JDK 16 - 2021)

> *“ We are improving and enriching the connections between the Java virtual machine and well-defined but "foreign" (non-Java) APIs, including many interfaces commonly used by C programmers.”*
>
> — Oracle, *Project Panama* [3]

[2] https://docs.oracle.com/javase/8/docs/technotes/guides/jni
[3] https://openjdk.java.net/projects/panama

In our previous project **neutrino**[4], we explored creating so called "*Proxy Objects*" for accessing structs in native space.



Using `sun.misc.Unsafe`[5], these proxy objects can write and read to their associated structs fields (in off-heap memory) directly.

[4] https://github.com/hhu-bsinfo/neutrino
[5] http://www.docjar.com/docs/api/sun/misc/Unsafe.html

Proxy objects create a mapping for struct fields using their names.

AddressHandle.java — Java

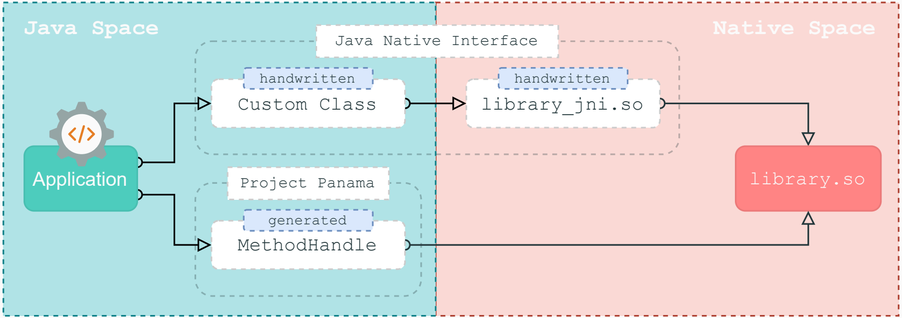
```java
@LinkNative("ibv_ah")
public class AddressHandle extends Struct {
  private final Context ctx = referenceField("context");
  private final ProtectionDomain pd = referenceField("pd");
  private final NativeInteger handle = integerField("handle");
}
```

verbs.h — C

```c
struct ibv_ah {
  struct ibv_context *context;
  struct ibv_pd *pd;
  uint32_t handle;
}
```

A lookup table must be managed **by hand** in native space.

The Java Native Interface requires handwritten "glue code" [6].



Project Panama automatically generates bindings through `jextract` [7]

[6] https://en.wikipedia.org/wiki/Glue_code
[7] https://github.com/openjdk/jextract

We provide a Gradle plugin [8] to automate `jextract`'s process

build.gradle (Groovy)

```groovy
plugins {
  id "io.github.krakowski.jextract"
   version "0.3.1"
}

jextract {
  header("${project
   .projectDir}/src/main/c/stdio.h")
   {
    libraries = [ 'stdc++' ]
    targetPackage = 'org.unix'
    className = 'Linux'
    functions = [ 'printf' ]
  }
}
```

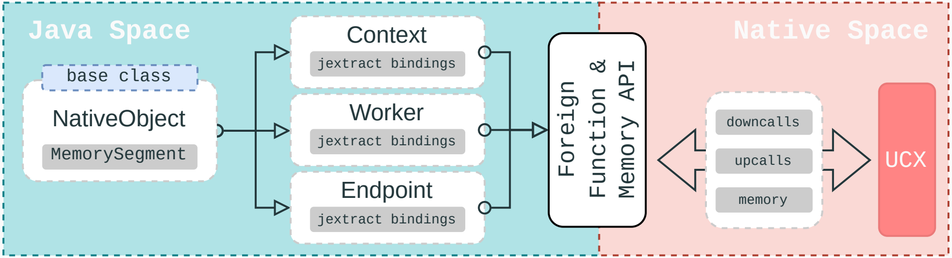
NativeHelloWorld.java (Java)

```java
import static org.unix.Linux.*;

public final class NativeHelloWorld {

  public static void main(String... args) {
    try (var session =
     MemorySession.openConfined()) {
      var format =
     session.allocateUtf8String("Hello %s");
      var value =
     session.allocateUtf8String("World");
      printf(format, value.address());
    }
  }
}
```

[8] https://github.com/krakowski/gradle-jextract

Framework Design

Using the **Foreign Function & Memory API**[9], we implement an object-oriented framework called **Infinileap**[10] wrapping the native `UCX` library.



[9] https://openjdk.org/jeps/424
[10] https://github.com/hhu-bsinfo/infinileap

NativeObject.java — Java

```java
public class NativeObject {
  private final MemorySegment segment;
  protected NativeObject(MemorySegment segment) { ... }
  protected NativeObject(MemoryAddress address, MemoryLayout layout) { ... }
  protected NativeObject(MemoryAddress address, long byteSize) { ... }
}
```

- All relevant UCX structs are wrapped inside a class having the same name.
- User **can** provide `MemorySession`

RequestParameters.java — Java

```java
public class RequestParameters extends NativeObject {
  public RequestParameters() {
    this(MemorySession.openImplicit());
  }

  public RequestParameters(MemorySession session) {
    super(ucp_request_param_t.allocate(session));
  }
}
```

Flags

UCX makes heavy use of flags within its provided structs.

Infinileap wraps these flags and encapsulates them in **enum values implementing a common interface**, so that they can be passed as **varargs** to our API.

LongFlag.java — Java

```java
@FunctionalInterface
public interface LongFlag {
  long getValue();
}
```

RequestParameters.java — Java

```java
public enum Field implements LongFlag {
CLIENT_ADDR(UCP_CONN_REQUEST_ATTR_FIELD_CLIENT_ADDR()),
CLIENT_ID(UCP_CONN_REQUEST_ATTR_FIELD_CLIENT_ID());

  private final long value;
  Field(int value) { this.value = value; }

  @Override
  public long getValue() { return value; }
}
```

Callbacks / Upcalls are provided as **functional interfaces**, which the user may implement. The framework calls `upcallStub` and stores the `MemorySegment` reference to prevent gargabe collection.

SendCallback.java — Java

```java
@FunctionalInterface
public interface SendCallback extends ucp_send_nbx_callback_t {
  void onRequestSent(long request, Status status, MemoryAddress data);

  @Override
  default void apply(MemoryAddress request, byte status, MemoryAddress data) {
    onRequestSent(request.toRawLongValue(), Status.of(status), data);
  }

  default MemorySegment upcallStub() {
    return ucp_send_nbx_callback_t.allocate(this, MemorySession.openImplicit());
  }
}
```

UCX enables atomic operations (e.g compare and swap) on remote memory. Infinileap abstracts this process by implementing so called "Native Primitives" which provide typed (`int`, `long`, ...) access to the underlying memory.

NativeLong.java — Java

```java
public final class NativeLong extends NativePrimitive {
  private static final int SIZE = Long.BYTES;
  public NativeLong() { this(MemorySession.openImplicit()); }
  public NativeLong(MemorySession session) { this(0, session); }
  public NativeLong(long initialValue, MemorySession session) {
    super(MemorySegment.allocateNative(SIZE, session), DataType.CONTIGUOUS_64_BIT);
    set(initialValue);
  }

  private NativeLong(MemorySegment segment) {super(segment, DataType.CONTIGUOUS_64_BIT);}
  public void set(long value) {segment().set(ValueLayout.JAVA_LONG, 0L, value);}
  public long get() {return segment().get(ValueLayout.JAVA_LONG, 0L);}
```

Native Primitives can be compared to the JDK's `Atomic`{`Integer`, `Long`, ...} classes.

AtomicAddExample.java

Java

```java
MemoryDescriptor descriptor = /* Received from other network participant */
RemoteKey remoteKey = endpoint.unpack(descriptor);

// Create a memory segment for atomic operations
var memorySegment = context.allocateMemory(Long.BYTES);
var nativeLong = NativeLong.map(memorySegment.segment());
nativeLong.set(32);

long request = endpoint.atomic(AtomicOperation.ADD, nativeLong,
    descriptor.remoteAddress(), remoteKey, new
    RequestParameters().setDataType(nativeLong.dataType()));

Requests.await(worker, request);
```

Messaging Example

Sender.java — Java

```java
final var buffer = MemorySegment.allocateNative(64L, MemorySession.openImplicit();

long request = endpoint.sendTagged(buffer);
Requests.await(worker, request); // busy-spin until request finishes
Requests.release(request); // clean up request
```

Receiver.java — Java

```java
final var buffer = MemorySegment.allocateNative(64L, MemorySession.openImplicit());

long request = worker.receiveTagged(buffer);
Requests.await(worker, request); // busy-spin until request finishes
Requests.release(request); // clean up request

MemoryUtil.dump(buffer, "Buffer"); // Print content
```

Full examples are available in our GitHub repository [11]

[11] https://github.com/hhu-bsinfo/infinileap/tree/master/example

Evaluation

Experiment Setup

CPU	Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz (22 MB Cache)
RAM	4x Micron Technology 36ASF2G72PZ-2G6E1 16GB
NIC	Mellanox Technologies MT27800 Family [ConnectX-5] (100Gbit/s)

Figure 1: System specifications of the hardware used in all experiments.

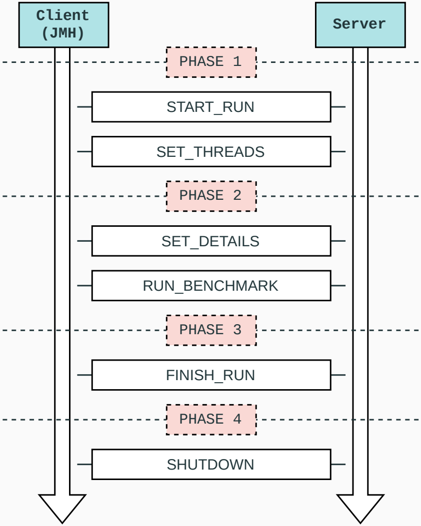
All benchmarks are implemented using the *Java Microbenchmark Harness (JMH)*[12] Framework. The benchmark's source code is available on GitHub[13]

All measurements shown are average values with standard deviation error bars

[12] https://github.com/openjdk/jmh
[13] https://github.com/hhu-bsinfo/infinileap

PHASE 1 Initialize the server instance and set the number of threads.

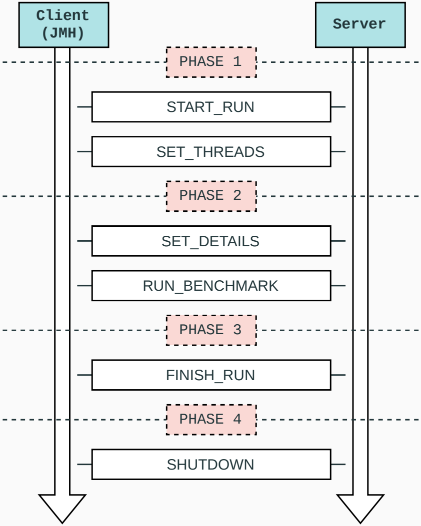
1. Send a `START_RUN` command to tell the server to start the next or first run.
2. Instruct the server to start a specified number of worker threads by sending a `SET_THREADS` command.



hhu.

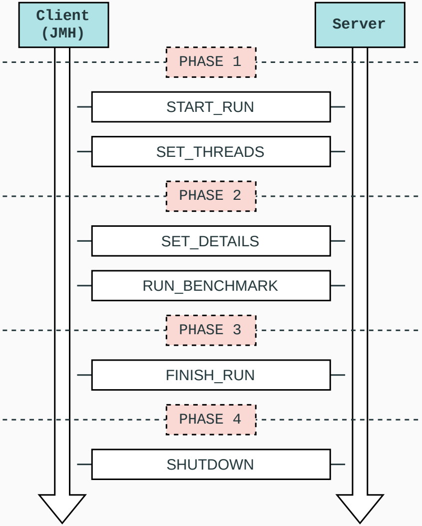
PHASE 2 JMH invokes benchmark methods and makes measurements.

1. Inform the server about the benchmark details (operation count, buffer size, etc.) by sending a `SET_DETAILS` command.
2. Execute the specified number of benchmark method invocations until a configured time has expired.
3. Synchronize with the server to let it receive new commands.
4. Repeat `PHASE 2` until all configured buffer sizes have been measured; else go to `PHASE 3`



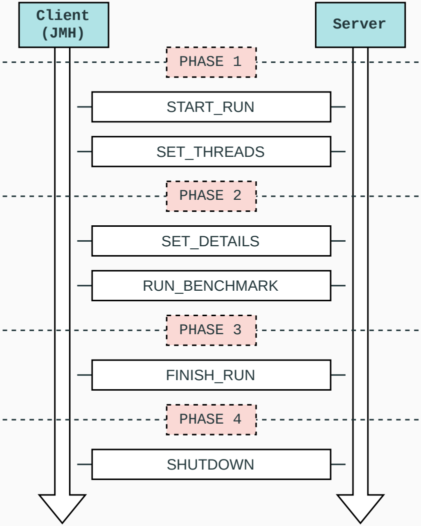
PHASE 3 - Resources are released.

1. Send a `FINISH_RUN` command to tell the server to release its resources and terminate all worker threads.
2. Start a new run with a different number of threads by reentering `PHASE 1` or move on to `PHASE 4`.



PHASE 4 - Benchmark ends.

1. Send a `SHUTDOWN` command to tell the server it should terminate. Alternatively another benchmark run may be started by sending a `START_RUN` command and starting again from the beginning.



hhu.

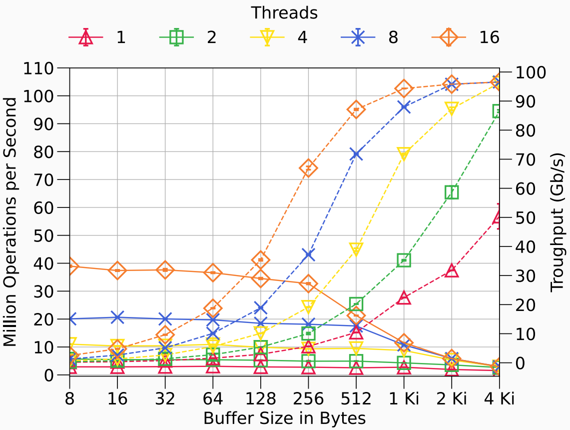


Figure 2: Average **read** operation (solid line) and network (dashed line) **throughput**

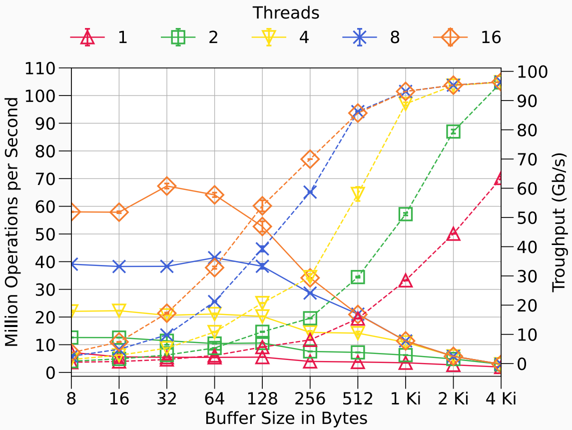


Figure 3: Average **write** operation (solid line) and network (dashed line) **throughput**

Results

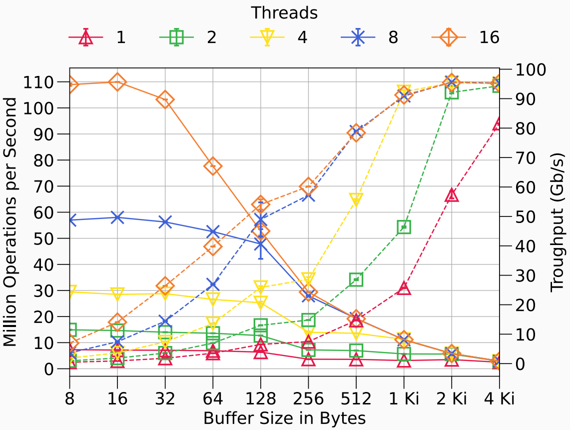


Figure 4: Average **send** operation (solid line) and network (dashed line) throughput

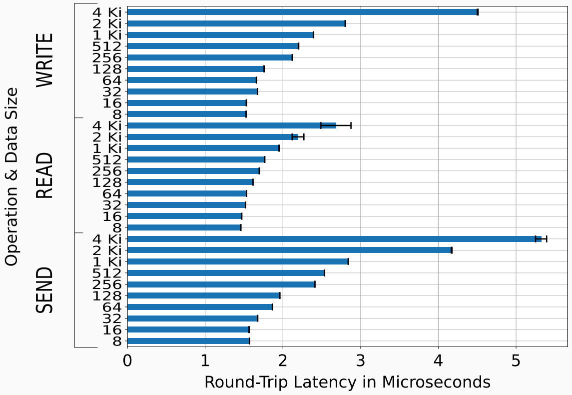


Figure 5: Average Round-trip latency for **RDMA write**, **RDMA read** and **send** operations

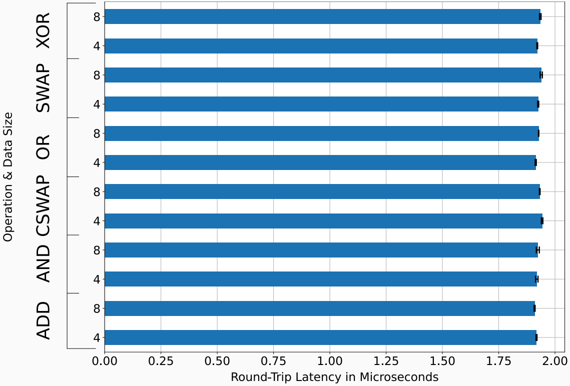


Figure 6: Average operation latency for all UCX-supported atomic operations and data sizes

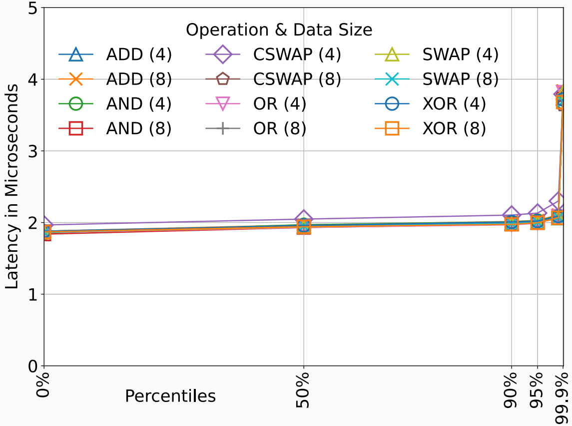


Figure 7: Operation latency for all supported atomic operations by percentiles

Conclusion & Lookout

- Project Panama's `jextract` tool enables easy integration of existing C libraries (direct call vs. glue code).
- Using the Foreign Function API in the "hot path" yields good results regarding performance.
- Thanks to the safety features of the Foreign Memory API, "off-by-one" errors are detected earlier in the development cycle.
- Higher-Level frameworks like Netty[14] can be accelerated using Infinileap.

[14] https://netty.io/

Questions