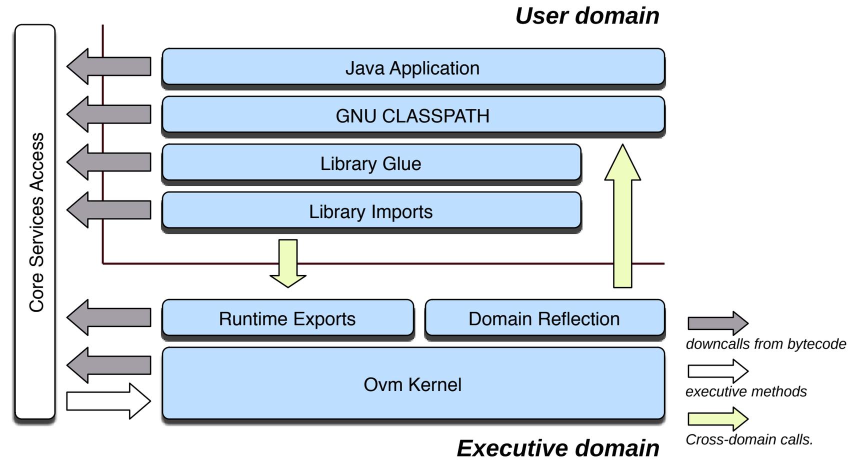
Ovm: a Real-time Java Virtual Machine for Avionics

The ScanEagle unmanned autonomous vehicle (UAV) with Boeing PRiSMj software and the Ovm Real-Time Java Virtual Machine. Having flown successfully and passed Boeing's internal qualification tests, the ScanEagle demonstrated the feasibility of using Real-Time Java in general, and the Ovm in particular in avionics applications. It was the first Real-Time Java system to do so. This poster describes the design of the Ovm's real-time components, with special focus on scheduling and scoped memory support. We also show the performance of the Ovm using a variety of benchmarks.

Basic Ovm architecture. The VM is split into the executive domain kernel, and the user domain, which contains the application and its libraries. Almost all of the Ovm is written in Java. All non-trivial Java bytecodes are converted to calls to *core services access* methods, which are implemented in the executive domain. The executive domain is also responsible for implementing scheduling, memory management, reflection, and I/O.



Operations that other VMs implement in native code are implemented in Java in the Ovm. This figure shows the code for allocation. Note that the code is syntactically Java, and gets compiled to Java bytecodes using an ordinary Java compiler.

```
VM_Address getMem(int size)
      throws PragmaNoPollcheck, PragmaNoBarriers {
   VM_Address ret = base().add(offset);
   offset += size;
   Mem.the().zero(ret.add(ALIGN), offset == rsize?size-ALIGN:size);
   return ret;
 }
```

However, the semantics of the code differ from Java—for example, the **throws** clause above is used to specify pragmas that alter the execution of this method. Further, the **VM_Address** class is *ephemeral*—it does not correspond to an object at run time; instead calls to it are translated into pointer manipulation operations.

```
static VM_Address* getMem(TransientArea* area, jint size){
  jint s1 = area + area->offset;
  area->offset += size;
  jint s2 = s1 + (&SplitRegionManager)->ALIGN;
  jint s3 = (area->offset == area->rsize)?
    (size-(&SplitRegionManager)->ALIGN) : size;
  PollingAware_zero(roots->values[57]), s2, s3);
  return s1;
}
```

All Java code in Ovm is executed via the J2c execution engine, which converts Java code to C++. In this code example, we see the **getMem** method seen previously converted to C++. We compile all Java methods to module-local C++ functions, allowing the C++ compiler to perform inlining. Most method calls are devirtualized—for example **Mem.the().zero()** is translated into the direct call **PollingAware.zero()**. Also, every **VM_Address** turns into an integer.

```
void someMethod() {

   ...
   while(...)  {
      ...

   }
}
```

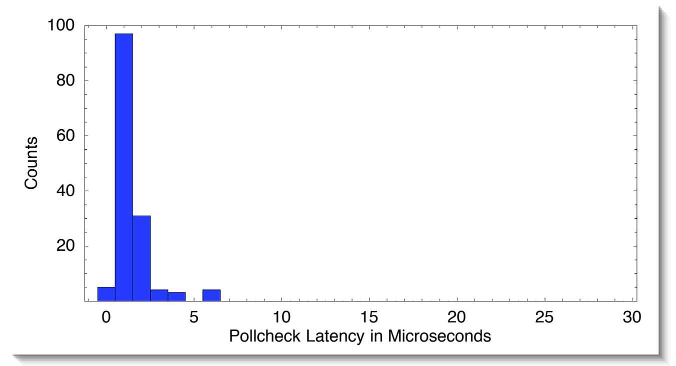
⇒

```
void someMethod() {
   POLLCHECK();
   ...
   while(...)  {
      ...
      POLLCHECK();
   }
}
```

Scheduling of threads is done without the help of the operating system. When code is compiled to Java, we insert *pollchecks*, which rapidly check if a scheduling decision needs to be made. Pollchecks are inserted at back branches (loops) and optionally at method entry, insuring the the number of instructions between pollchecks is bounded. When a pollcheck fires, the event handling and thread management framework decides which thread to run next.

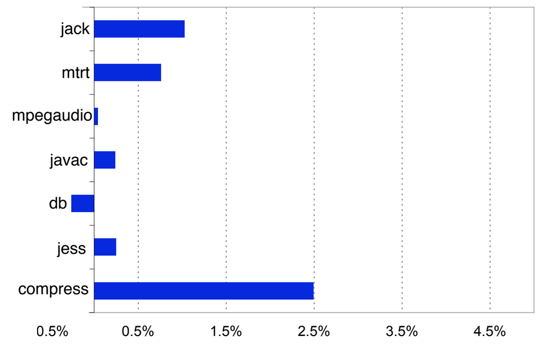
```
union {
  struct {
   volatile int16_t notSignaled;
   volatile int16_t notEnabled;
  } s;
  volatile int32_t pollWord;
} pollUnion;


POLLCHECK:
 if (pollUnion.pollWord == 0) {
   pollUnion.s.notSignaled = 1;
   pollUnion.s.notEnabled = 1;
   handleEvents();
 }
```

Pollchecks are fast to execute, and fast to disable. A pollcheck simultaneously checks for two flags: *signaled* and *enabled*. The signaled flag is set asynchronously by interrupt handlers written in C that detect conditions that would require rescheduling (such as a timer interrupt). The enabled flag specifies if pollchecks are enabled (clearing this flag enables atomic execution). The logic is set up to allow the fastest possible pollcheck without having to use atomic instructions.

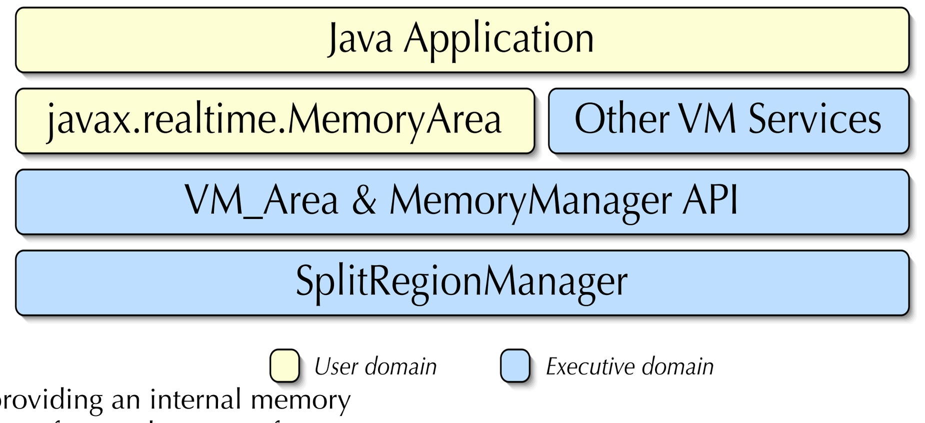


One of the concerns of using a pollcheck scheme for scheduling is the time between pollcheck executions. If this latency is too great, scheduling decisions may come too infrequently. This histogram shows the pollcheck latency in microseconds. The worst case is about 6 microseconds.

Throughput is also a concern. How much slower is Ovm code with pollchecks included? This figure seeks to answer this question. We compare the execution time of various SpecJVM98 benchmarks with and without pollchecks. Notice that the worst-case overhead is around 2.5%. We see that although pollchecks require code to be added to every method, it does not significantly impact performance.



Real-Time Java relies on the scoped memory API to guarantee that high-priority tasks can execute without garbage collector interference. The **javax.realtime.MemoryArea** class serves as the parent class of the scoped memory area class hierarchy. Since the Ovm is written in Java, all virtual machine functions also need to be written in such a way as to avoid collector interference. We do this by providing an internal memory management API that contains a superset of scoped memory features.




```
public ValueUnion call(Oop recv) {
    VM_Area area = MemoryManager.the().getCurrentArea();
    Object r1 = MemoryPolicy.the().enterScratchPadArea();
    try {
        InvocationMessage msg = makeMessage();
        VM_Area r2 = MemoryManager.the().setCurrentArea(area);
        try {
            ReturnMessage ret = msg.invoke(recv);
            ret.rethrowWildcard();
            return ret.getReturnValue();
        } finally { MemoryManager.the().setCurrentArea(r2); }
    } finally { MemoryPolicy.the().leave(r1); }
}
```

One of the features that the Ovm memory management API adds is the *scratch pad*, a memory area that provides functionality similar to **alloca** in C. This is a recursive area—exiting it reclaims only those objects that were allocated since the most recent entry. This allows us to allocate temporary objects without having to find the appropriate scope. In this code example, we see a method that implements reflective calls in the executive domain. Because it needs to allocate the temporary **InvocationMessage** object, we enter into the scratch pad using our **MemoryPolicy** and **MemoryManager** APIs. If we had used the real-time Java scoped memory API, the code would have to contain complicated logic for finding or allocating the appropriate scope—something that is never necessary in Ovm.

```
Vector::
void ensureCapacity(int cap){
   ...
  Object[] arr=(Object[]) thisArea.newArray(Object.class, cap);
  System.arraycopy(elementData,0,arr,0,elementCount);
  elementData=arr;
}
```

User domain code also requires special care in the presence of scoped memory. Here, we see a modified **java.util.Vector** method. This method has been changed so that calls from outside the vector object's parent scope do not cause memory errors. In this example, **thisArea** is the **MemoryArea** of the receiver. The **newArray()** method is used to reflectively allocate the new backing store of the **Vector**.

```
void readBarrier(VM_Address src)
      throws PragmaInline, PragmaNoBarriers, PragmaNoPollcheck {
   if (!doLoadCheck)  return;
   if (src.diff(heapBase).uLessThan(heapSize))  fail();
 }
```

Real-time Java guarantees that high priority threads do not have to deal with garbage collector interruptions. This guarantee is enforced by a *read barrier* that the compiler inserts before every heap read operation. Its job is to verify that the heap is not accessed by threads that may preempt the collector. The code for the Ovm read barrier is shown in this figure. Our read barrier is fast—we simply perform arithmetic on the target object's address to insure that it does not fall outside of the heap.

```
void storeCheck(VM_Address src, int offset, VM_Address tgt)
     throws PragmaNoPollcheck, PragmaNoBarriers, PragmaInline  {
     int sb = src.asInt() >>> blockShift;
     int tb = tgt.asInt() >>> blockShift;
     if (sb != tb) storeCheckSlow(sb, tb);
 }
```

Writes to memory also need to be checked in real-time Java, to insure that longer-lived objects never point at shorter-lived ones. Object lifetime is determined by the object's scoped memory area. A write barrier is used to perform this check. The Ovm store check fast path is shown in this figure. The fast path simply verifies that the objects are in the same page.

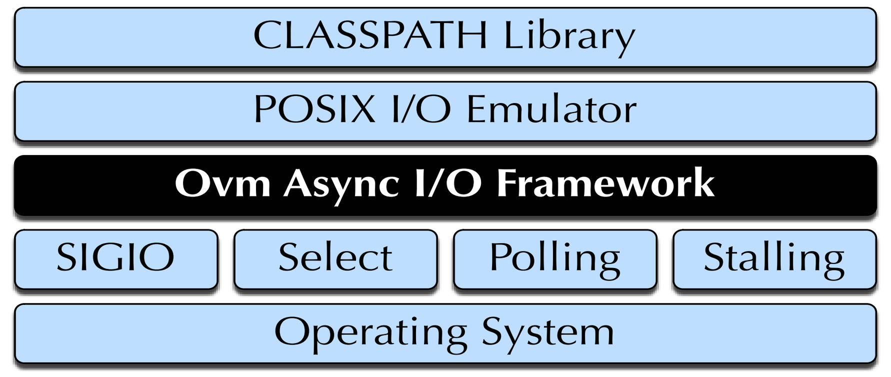
```
void storeCheckSlow(int sb, int tb)
   throws PragmaNoPollcheck, PragmaNoBarriers, PargamNoInline {
   VM_Word tidx = VM_Word.fromInt(tb - scopeBaseIndex);
   if (!tidx.uLessThan(scopeBlocks)) return;
   Area ta = scopeOwner[ tidx.asInt() ];
   VM_Word sidx = VM_Word.fromInt(sb - scopeBaseIndex);
   if (!sidx.uLessThan(scopeBlocks)) fail();
   Area sa = scopeOwner[sidx.asInt()];
   if (sa == ta)  return;
   if ((ta.prange - sa.crange) & MASK) != RES) fail();
 }
```

The slow path of the store check is shown here. In Ovm, store checks are always *O(1)* in time and space. For a detailed description of the algorithm see Palacz and Vitek's *Subtype tests in real time.*

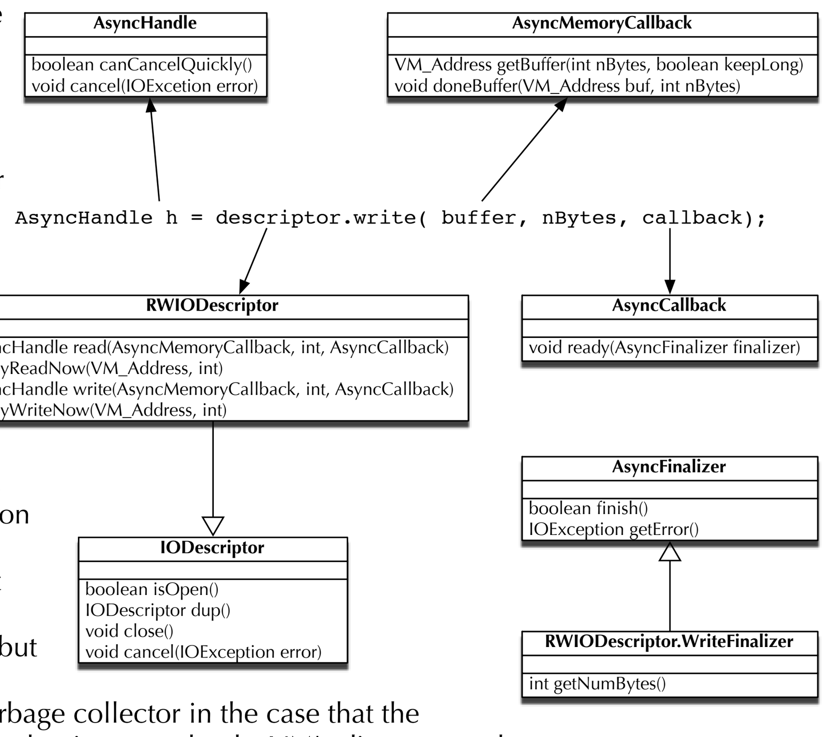
```
VM_Area areaOf(Oop mem) {
  VM_Word off = VM_Address.fromObject(mem).diff(heapBase);
  if (off.uLT(VM_Word.fromInt(heapSize))) return heapArea;
  off = VM_Address.fromObject(mem).diff(scopeBase);
  if (!off.uLT(VM_Word.fromInt(scopeSize))) return immortalArea;
  int idx = off.asInt() >>> blockShift;
  return scopeOwner[idx];
}
```

In Ovm, finding the memory area that owns an object is fast and does not require an extra header field in the object. We simply maintain a page-to-memory-area mapping (see the **scopeOwner** array). To find the memory area of an object, we first round down the object's base address to the base of the page, and then look up the memory area associated with the page. The process is fast and reduces memory usage by eliminating the need for an extra field in the object header.

Ovm manages its own scheduling. This means that the I/O scheduling machinery found in the operating system kernel must be duplicated in the virtual machine, to insure that the whole VM does not block on a single thread's I/O operation. In this figure we see an illustration of the Ovm I/O stack. At the top is GNU CLASSPATH, which is our implementation of the Java class libraries. CLASSPATH expects to be able to use a POSIX I/O interface—so we provide it, using our POSIX I/O emulator. The scheduling is managed by the Ovm Async I/O Framework, seen in black. The POSIX I/O emulator, which provides blocking I/O operations, is implemented in terms of the asynchronous operations provided by Ovm Async I/O. In turn, the Async I/O framework has multiple implementations, ranging from the conservative Polling implementation (intended to work on any device) to the high-throughput select implementation. SIGIO is the implementation we use most frequently for real-time application. The the following figures we describe the Async I/O framework in detail.

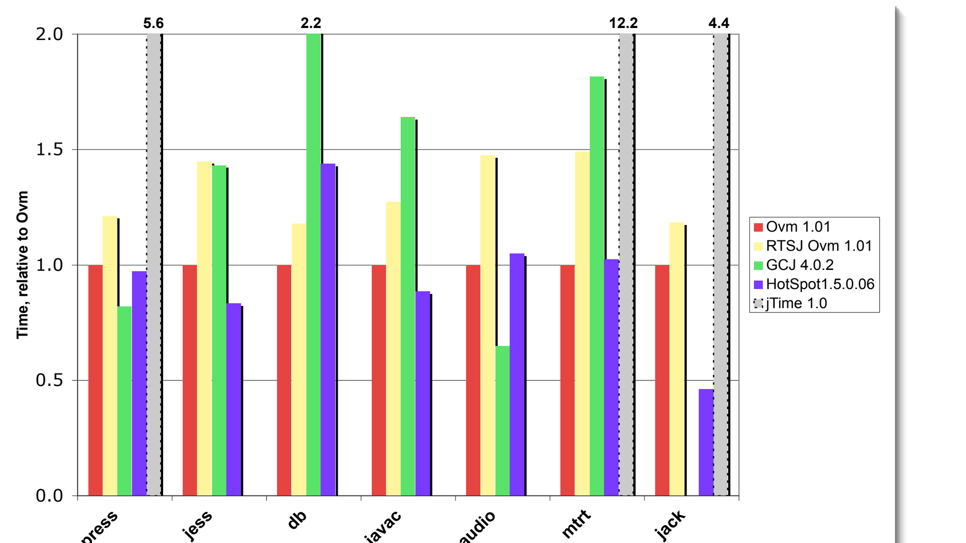


Anatomy of an Async I/O call. We use the **write** operation as a running example. Operations are meant to look like their POSIX counterparts with the exception that they are designed to return immediately, rather than upon completion of the operation; and in that instead of an integer file descriptor, we have an **IODescriptor** object. Being asynchronous, every operation requires a callback that's used for notifying the client when the operation completes. Additionally, a handle is returned that allows the client to cancel the operation after it is initiated. Perhaps most strikingly, the async operations do not accept regular memory buffers (in the form of a pointer, or an array object), but instead require a special callback, to insure optimal interaction with the garbage collector in the case that the operation is implemented by a process that is not under the VM's direct control.




```
public int write(
        int fd, Oop buf, int byteOffset, int byteCount, boolean block) {
    if (byteCount == 0) return 0;
    if (!verifyPointer(buf, byteOffset, byteCount)) {
        setErrno(NativeConstants.EFAULT);
        return -1;
    }
    IODescriptor io = getIOD(fd);
    if (io == null) {
        setErrno(NativeConstants.EBADF);
        return -1;
    }
    if (!(io instanceof RWIODescriptor)) {
        setErrno(NativeConstants.EINVAL);
        return -1;
    }
    // (a) Non-blocking fast path
    try {
        int result = ((RWIODescriptor)io).tryWriteNow(
                getPointer(buf, byteOffset, byteCount),
                byteCount);
        if (result >= 0) return result;
    } catch (IOException e) {
        setErrno(e);
        return -1;
    }
    if (!block) {
        setErrno(NativeConstants.EWOULDBLOCK);
        return -1;
    }
    // (b) Slow path that uses asynchronous I/O
    Object r1 = MemoryPolicy.the().enterScratchPadArea();
    try {
        BlockingCallback bc = new BlockingCallback(bm,tm);
        ((RWIODescriptor)io).write(
                new ForWriteMemoryCallback(buf,byteOffset,byteCount),
                byteCount, bc);
        bc.waitOnDone();
        IOException error = bc.getFinalizer().getError();
        if (error != null) {
            setErrno(error);
            return -1;
        }
        return ((RWIODescriptor.WriteFinalizer)bc.getFinalizer()).getNumBytes();
    } finally {
        MemoryPolicy.the().leave(r1);
    }
}
```

POSIX I/O implementation of the **write** operation. We proceed much as a typical operating system would: after performing sanity checks, in (a) we attempt a non-blocking fast path. If the operation is configured to be blocking (see the **block** parameter), the code drops to the slow path in (b), where the Async I/O **write** operation is used to emulate POSIX blocking semantics. Other I/O operations are implemented in a similar fashion.

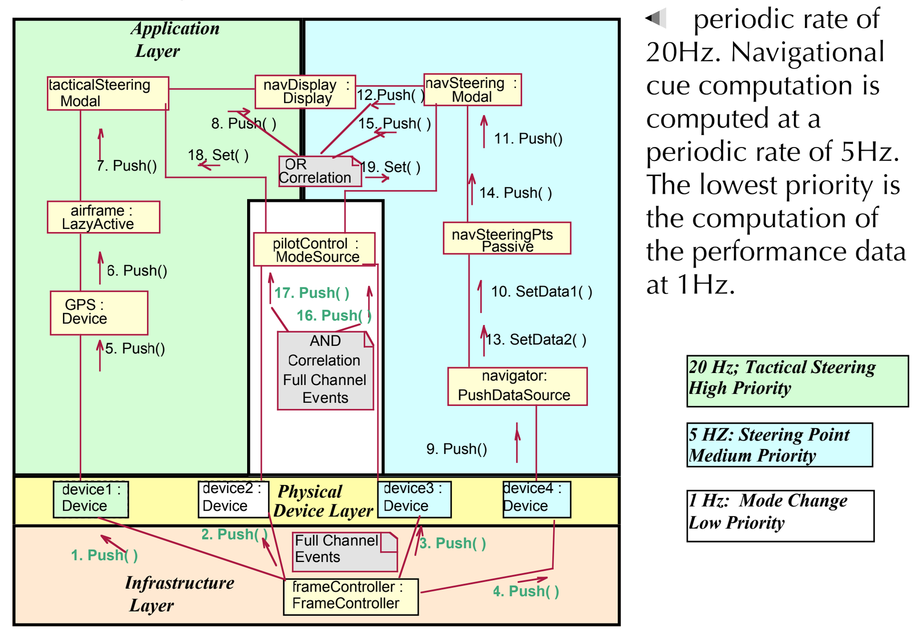


Ovm performance compared to a number of other VMs, including those optimized for throughput (like HotSpot) and for real-time (like jTime). Lower numbers are better. Note that we consider both real-time and throughput configurations of the Ovm. GCJ and jTime were unable to complete a number of benchmarks. In all cases where jTime completed the benchmark, it ran for much longer than Ovm. Ovm was the fastest RTJVM that we were able to test, and its performance tended to be in the same ballpark as HotSpot.

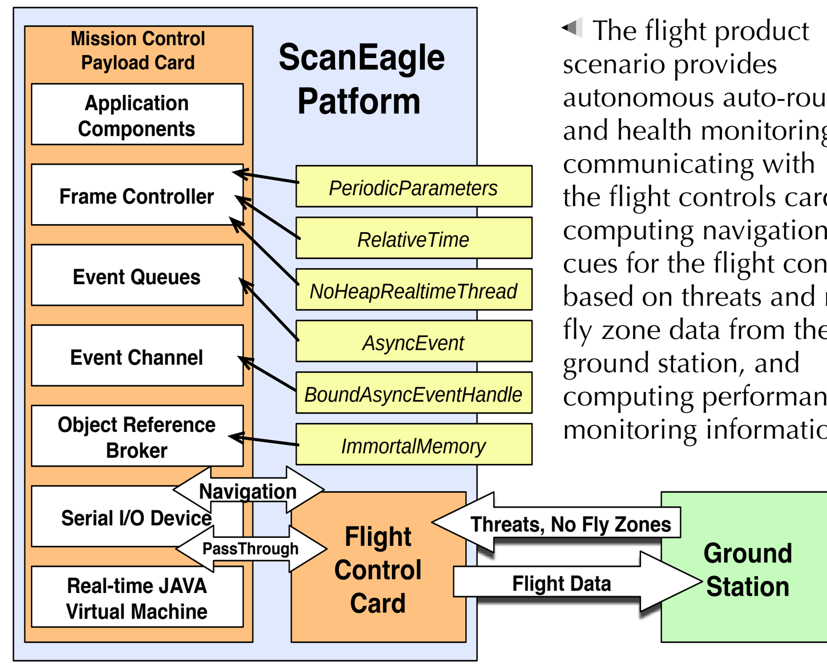
	LOC	Classes	Data	Code
Boeing PRiSMj	108’004	393	22’944 KB	11’396 KB
UCI RT-Zen	202’820	2447	26’847 KB	12’331 KB
GNU classpath	479’720	1974	–	–
Ovm framework	219’889	2618	–	–
RTSJ libraries	28’140	268	–	–

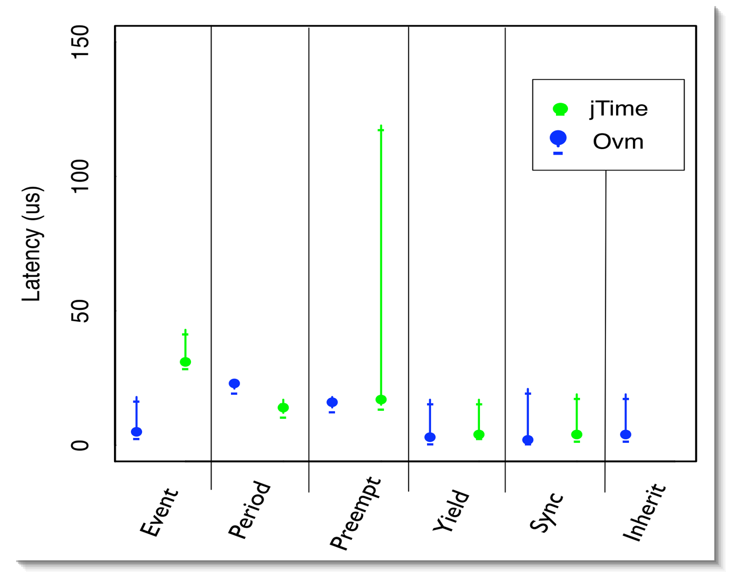
Size of the Ovm, associated libraries, and two applications that we use. The Ovm itself consists of just over 200,000 lines of code. The implementation of the RTSJ itself is quite small, but don't be fooled—the RTSJ libraries make heavy use of Ovm framework functionality that would not be there if we did not support the RTSJ. The GNU CLASSPATH library is considerably larger than the Ovm. PRiSMj, the ScanEagle application, and the UCI RT-Zen ORB are two applications that we run. Both are over 100,000 lines of code.

Overview of the PRiSMj application. Synchronized communication with the flight controls is a mission critical function and executed at a periodic rate of 20Hz. Navigational cue computation is computed at a periodic rate of 5Hz. The lowest priority is the computation of the performance data at 1Hz.

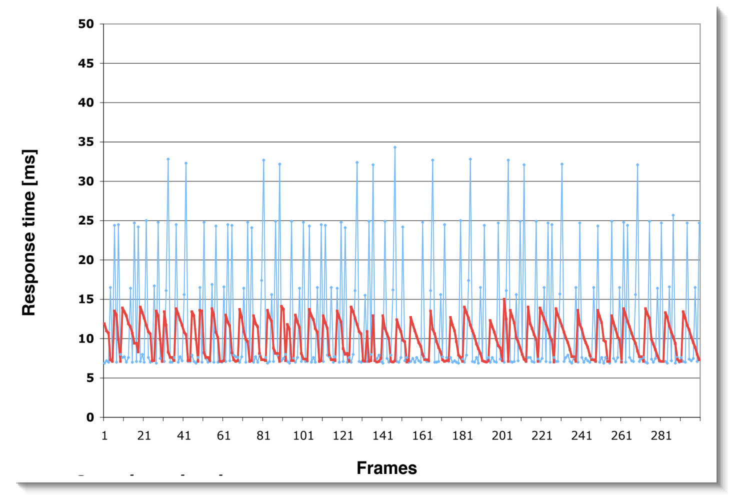


The flight product scenario provides autonomous auto-routing and health monitoring by communicating with the flight controls card, computing navigational cues for the flight controls based on threats and no fly zone data from the ground station, and computing performance monitoring information.

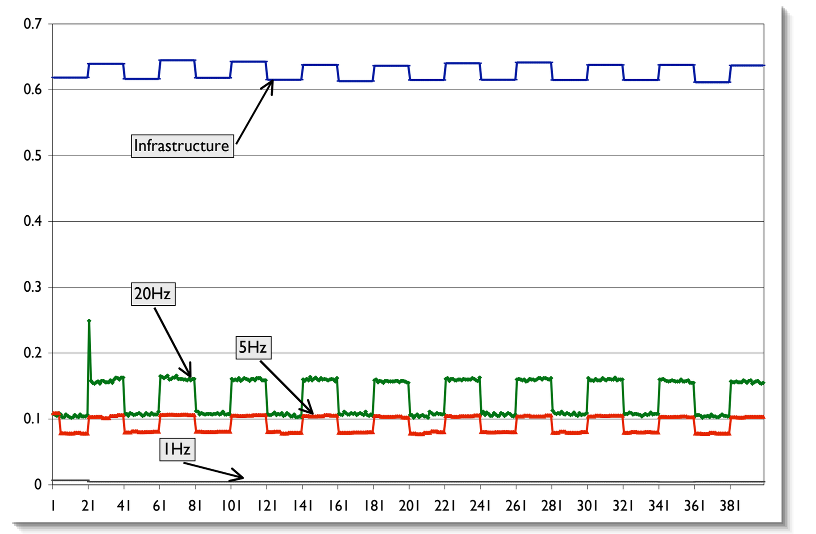




Real-time performance microbenchmarks. Lower numbers are better. In all benchmarks except for Period, Ovm is at least as good as jTime. In the Inherit benchmark, which measures the performance of priority inheritance locks, jTime was unable to complete the test.



Predictability of RT-Zen running on the Ovm. We have two thread groups, low-priority and high-priority, handling 300 requests each. Of note is that we do not see any major outliers in request processing time.



The Ovm implementation of PRiSMj was the first application to qualify to fly on the ScanEagle UAV. Performance of PRiSMj on Ovm is shown in this figure. Response times of 100 threads split in three groups (high, medium, low) on a modal workload are shown. The x-axis shows the number of data frames received by the UAV control, the y-axis indicates the time taken by by a thread to process the frame in milliseconds. Our jitter is well within the 1% jitter target.

Research by Jason Baker, Antonio Cunei, Chapman Flack, Filip Pizlo and Jan Vitek of Purdue University;
Austin Armbuster and Edward Pla of the Boeing Company;
David Holmes of DLTeCH; and
Marek Prochazka of SciSys.