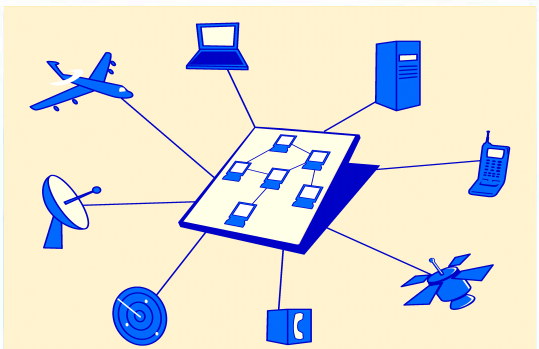


The Impact of a Real-Time JVM on Middleware Performance: Lessons Learned from Implementing DDS on IBM's J9

Ken Brophy, Senior Applications Engineer, RTI
Rick Warren, Lead Software Engineer, RTI

Agenda

This Presentation Has Two Major Sections

- Latency Performance Comparisons
- RTSJ Observations

Latency Performance Measurements

Performance measures include:

- RTSJ-enabled JVM using RealtimeThread
  - without real-time enabled (i.e. ‘normal’ mode)
  - with real-time enabled but without RealtimeThreads
  - with real-time enabled and using RTSJ features
- ‘Standard’ JVM
  - This is a Sun 1.5 JVM

Configuration

RHEL 4

```
Linux host 2.6.16-rtj12.9.1smp #1 SMP PREEMPT Wed Sep 27 15:05:54 PDT 2006 i686
athlon i386 GNU/Linux
```

IBM J9 JVM

```
host [162] /opt/java2/jre/bin/java -version
java version "1.5.0"
Java(TM) 2 Runtime Environment, Standard Edition (build pxi32rt23-20060824a)
IBM J9 VM (build 2.3, J2RE 1.5.0 IBM J9 2.3 Linux x86-32 j9vmxi3223ifx-20060719
(JIT enabled)
J9VM - 20060714_07194_lHdSMR
JIT  - 20060428_1800.ifix2_r8
GC   - 200607_07)
JCL  - 20060816
host [163] /opt/java2/jre/bin/java -fullversion
java full version "J2RE 1.5.0 IBM Linux build pxi32rt23-20060824a“
```

Sun JVM

```
host [164] java -version
java version "1.5.0_07"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_07-b03)
Java HotSpot(TM) Server VM (build 1.5.0_07-b03, mixed mode)
host [165] java -fullversion
java full version "1.5.0_07-b03“
```

Processors & Network

```
Dual CPUs with dual cores @ 2GHz
High Performance GB Router
```

Agenda

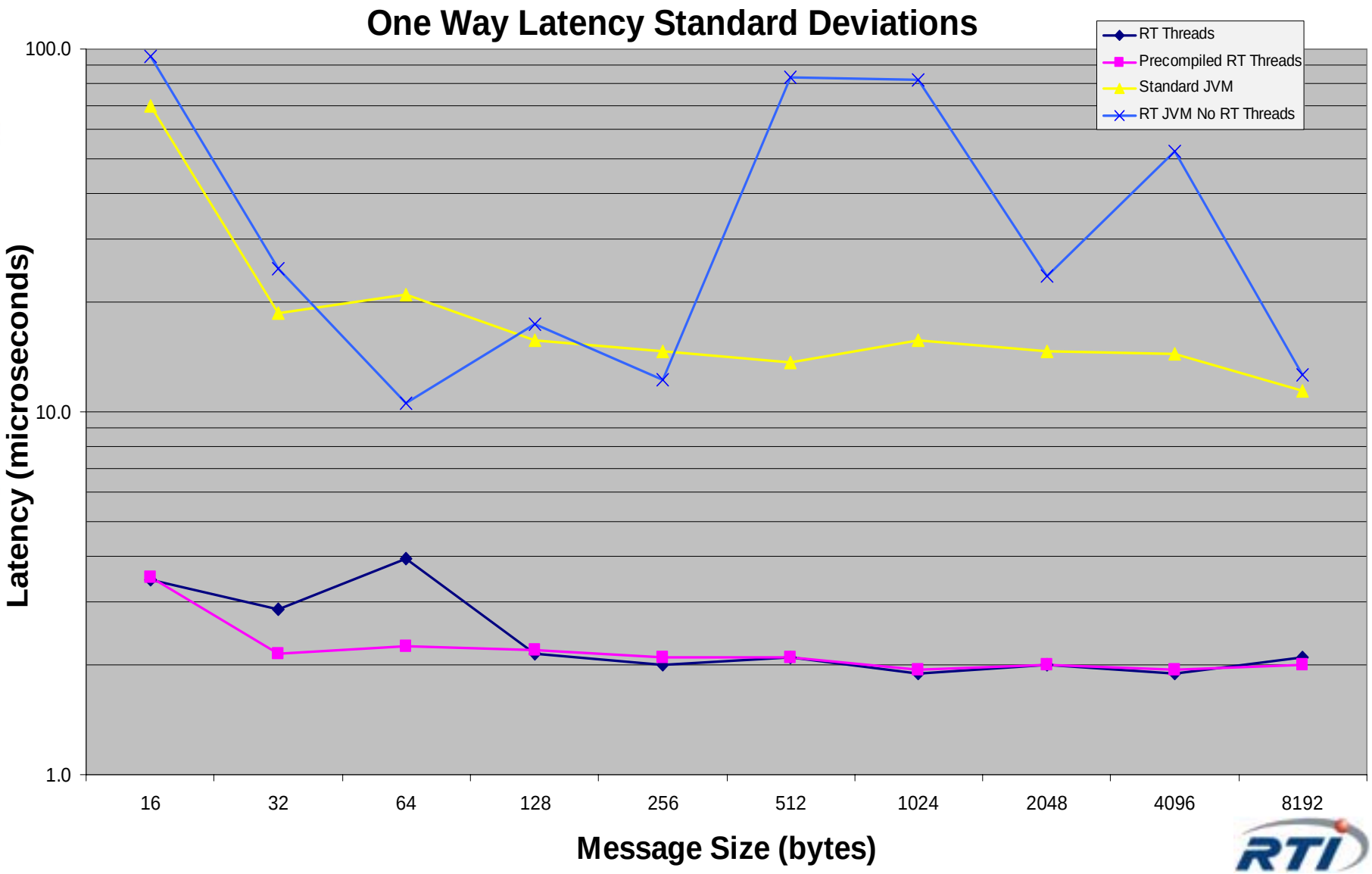
This Presentation Has Two Major Sections

- **Latency Performance Comparisons**
- RTSJ Observations

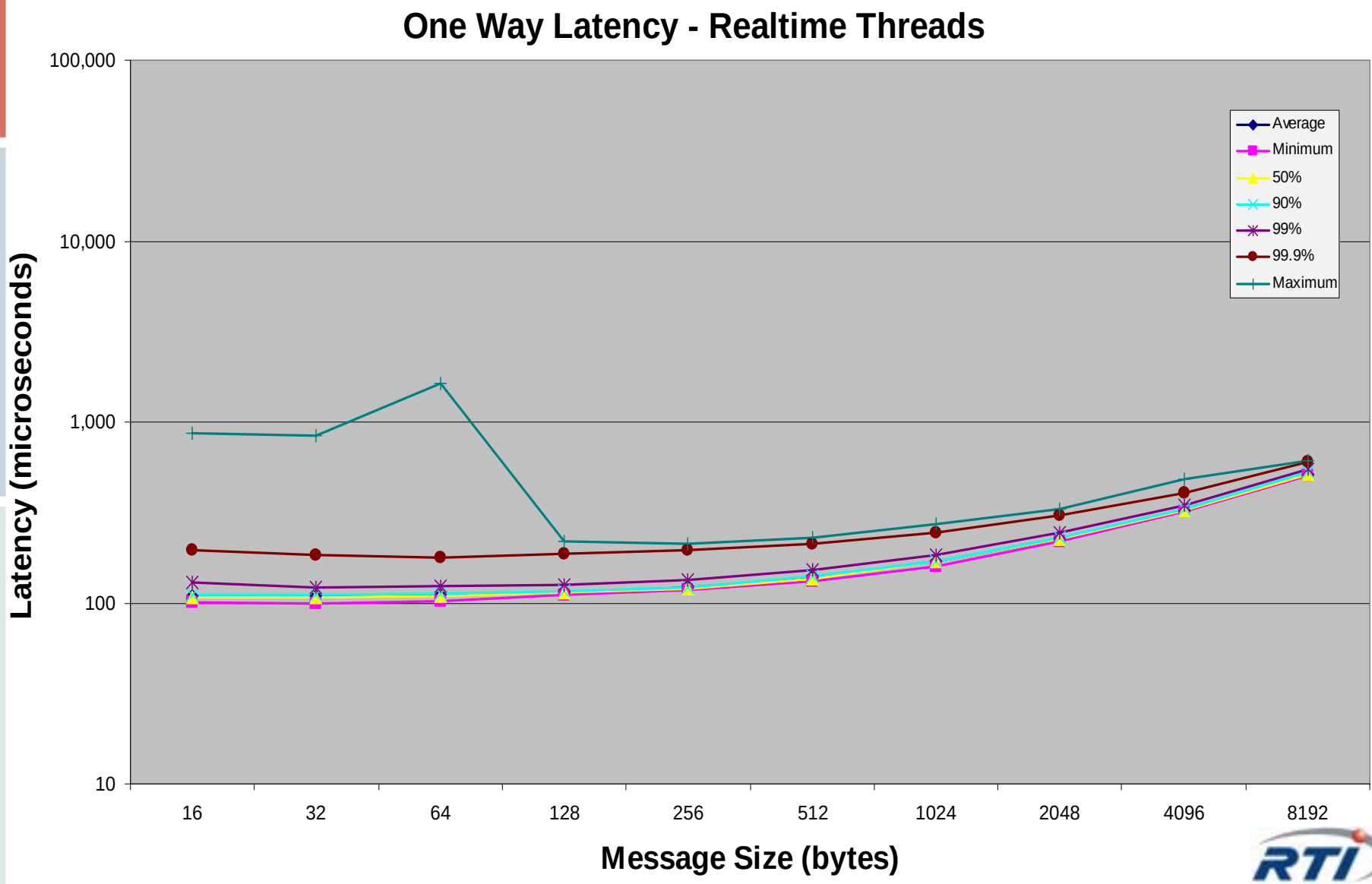
Conclusions

- The IBM J9 RTSJ JVM provides much improved determinism over standard JVMs
  - 1024 byte message one way latency standard deviation results
    - 2µs for RTSJ JVM with RealtimeThread
    - 16µs for Standard JVM
- This determinism does come at a cost, however. Average performance is not quite as good…
  - 1024 byte message one way latency results
    - 170µs for RTSJ JVM with RealtimeThread
    - 107µs for Standard JVM

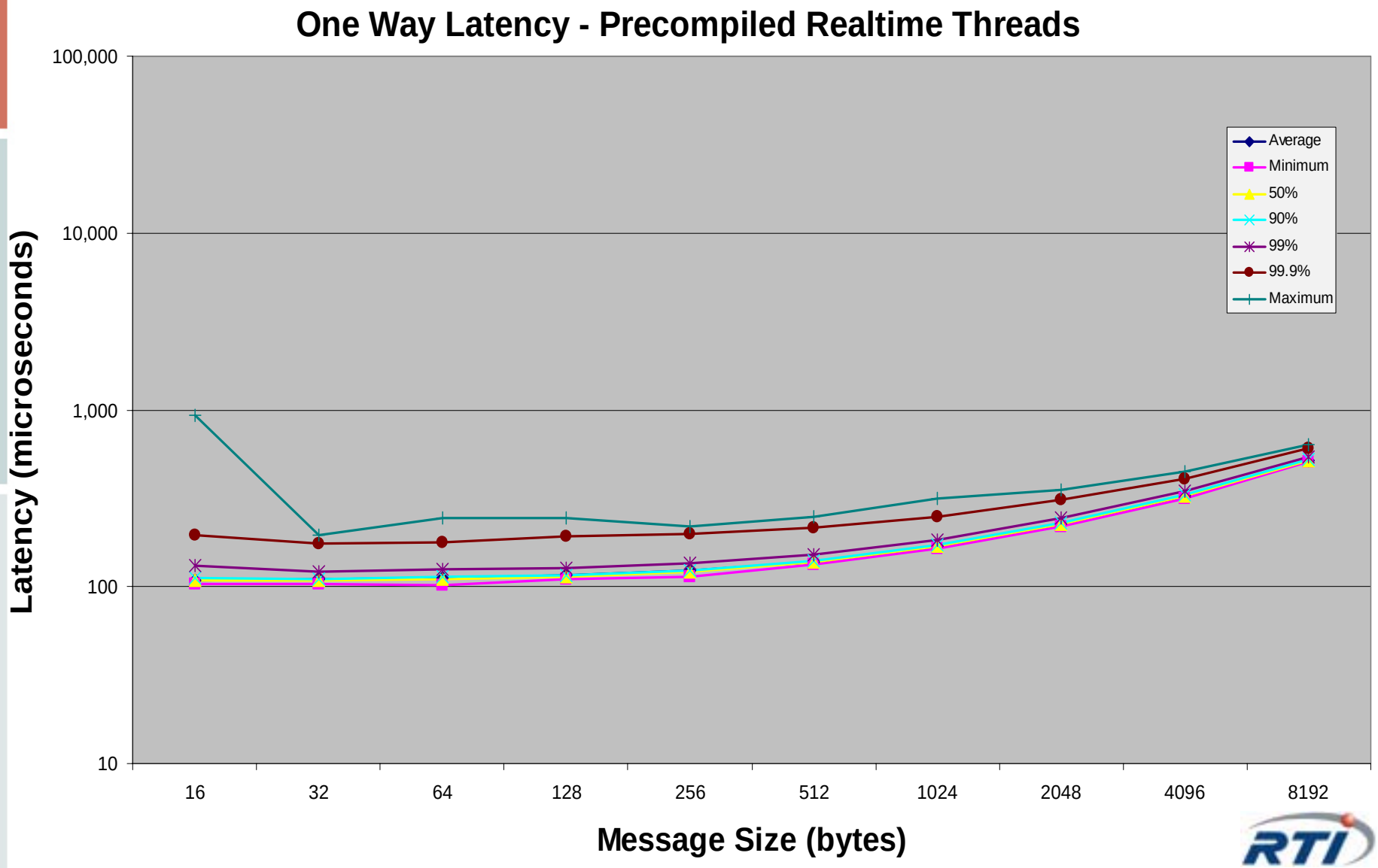
Latency Standard Deviations



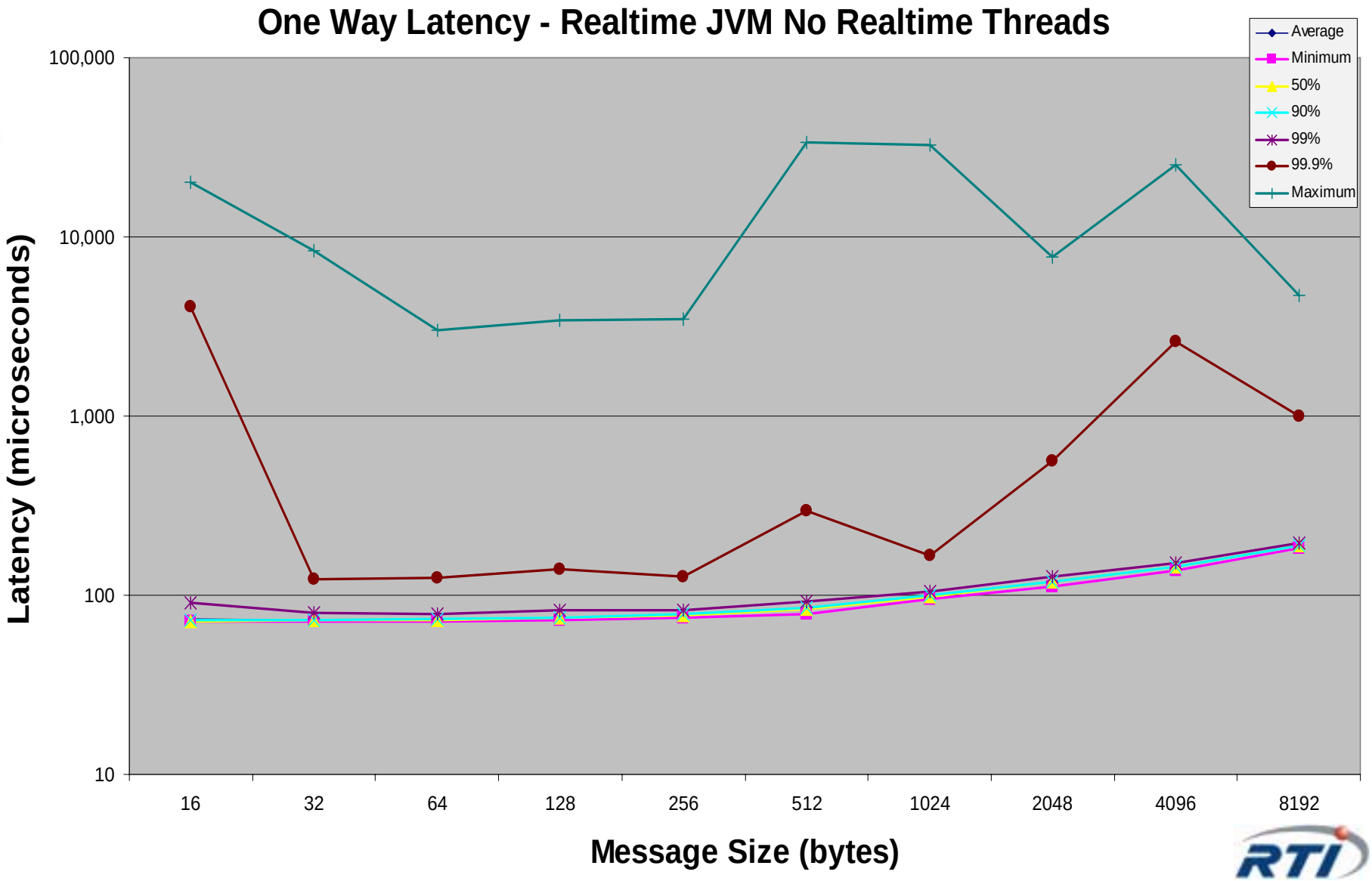
Latency – Realtime Threads



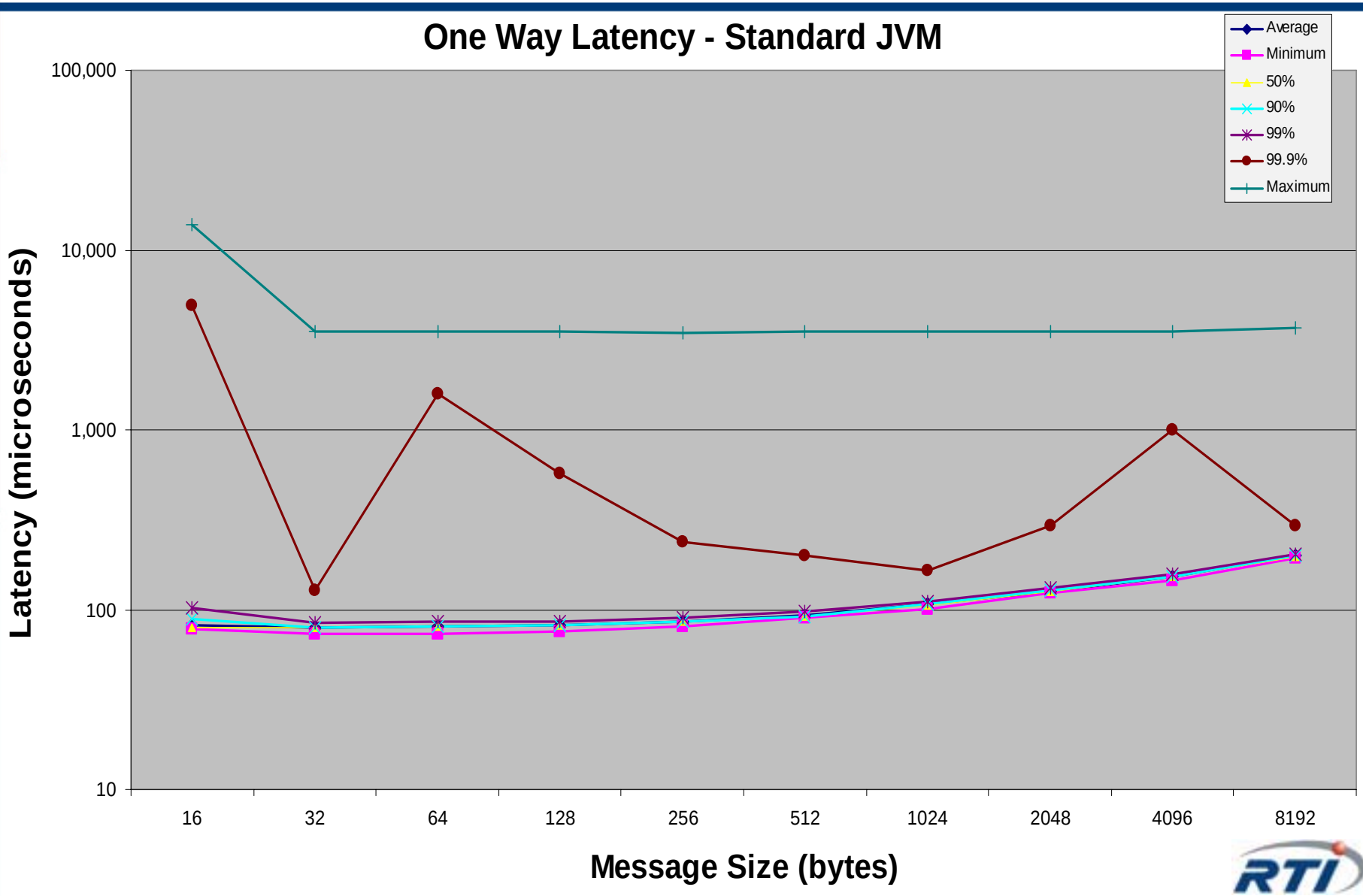
Latency – Precompiled Realtime Threads



Latency – RT JVM No Realtime Threads



Latency – Standard JVM



Agenda

This Presentation Has Two Major Sections

- Latency Performance Comparisons
- **RTSJ Observations**

Why Real-Time Java?

- High, deterministic performance
  - Controllable, predictable garbage collection/ object disposal
  - Controllable, predictable thread scheduling
  - Minimal latency penalty for high-level language features
- Ease of use
  - Take advantage of modern language and rich class library
    - …to develop faster
    - …with fewer errors

Ease of Use?

Control over memory allocation / de-allocation comes at a high price

- Lack of portability
  - Standard libraries ***might*** work
  - Standard language features ***might*** work
  - Thread priorities aren't standardized
- Complicates design and implementation
  - Complex memory access rules
  - No compile-time enforcement
  - **`Runnable`**s within **`Runnable`**s within **`Runnable`**s

RTI

Lack of Standard Library Support

- Java standard libraries may not work in non-GC memory areas
  - Not all **`java.*`** packages supported — to be expected
    - Including **`java.lang.*`**
    - Including **`java.lang.Object.*()`**
  - Which APIs are supported?
    - Not well documented
    - Varies by VM vendor
- *Request to vendors*: Give us something we can count on

Lack of Support for Language Features

The Java 5 language itself is not entirely RTSJ-compatible

- *Pretty obvious*: enhanced **`for`** loop
  - **`for (`*`variable`*` : `*`collection`*`)`**
  - Allocates iterator: can't use in non-GC memory
- *Somewhat less obvious*: boxing primitives
  - Passing primitives to methods expecting objects
  - *May* allocate wrapper, or may reuse pre-allocated
    - Depends on primitive type and value
    - Read Java Language Specification to learn more
- *Not obvious*: switching on enums
  - **`switch (`*`variable`*`) { case `*`ENUM_CONSTANT`*`: }`**
  - Static fields added to byte-code cause memory access errors

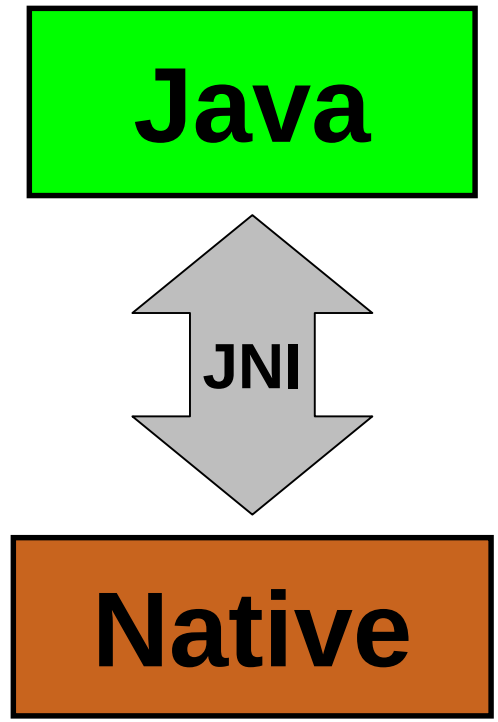
RTI

RTSJ and JNI

- Flavors that go well together?
  - JNI: The Other Way to call real-time code from Java
- *Good news*: Few restrictions
  - Any thread can call native code
  - Any thread can allocate memory on the native heap
- *Bad news*: Inconsistent threading support
  - Real-time threads must be created in Java code
    - No standard JNI API to spawn real-time threads
    - No way to **`attach`** native thread to JVM as real-time thread

RTSJ and JNI (*cont.*)

	Non-Real-Time	Real-Time
Java Thread	● Java → Native ● Native → Java	● Java → Native ● Native → Java
Native Thread	● Java → Native ● Native → Java	● Java → Native ● ~~Native → Java~~



Real-Time is Hard

- Designing a hard real-time application is complex in any language
  - "Java is easier to program in than C, and RTSJ is based on Java…" *–careful!*
- Memory access rules are complex and easy to get wrong
  - No compile-time checking
  - *Example gotcha*: class initialization runs in immortal memory
    - Therefore all Class objects are in immortal memory
    - Therefore all static data must also (recursively) be in immortal memory

Some Friendly Advice

- Real-time JVM brings real benefits
  - Whether or not you use RTSJ
- Evaluate the need for RTSJ very carefully
  - If you know regular Java won't do the job, consider native code
    - C++ is no harder to learn than RTSJ
    - JNI or language-neutral middleware can bridge the gap
- If you do use RTSJ…
  - Developers need training and experience
  - Test, test, test
    - Fatal memory access errors can only be caught at runtime
    - Fixing them *will* impact your design
    - Tests *must* exercise real-world threading and memory access patterns to be valid
    - JUnit is of limited help: depends on GC