Jfokus 2020
HotSpot Handshaking

Robbin Ehn

JPG – HotSpot Runtime
February 3, 2020
robbin.ehn@oracle.com

Safe Harbor

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, timing, and pricing of any features or functionality described for Oracle's products may change and remains at the sole discretion of Oracle Corporation.

Statements in this presentation relating to Oracle's future plans, expectations, beliefs, intentions and prospects are "forward-looking statements" and are subject to material risks and uncertainties. A detailed discussion of these factors and other risks that affect our business is contained in Oracle's Securities and Exchange Commission (SEC) filings, including our most recent reports on Form 10-K and Form 10-Q under the heading "Risk Factors." These filings are available on the SEC's website or on Oracle's website at http://www.oracle.com/investor. All information in this presentation is current as of September 2019 and Oracle undertakes no duty to update any statement in light of new information or future events.

Intro

Handshakes - Latency and throughtput?

- Low latency GC's
- Runtime
- Handshake
- Effective
- Limited

HotSpot Handshaking

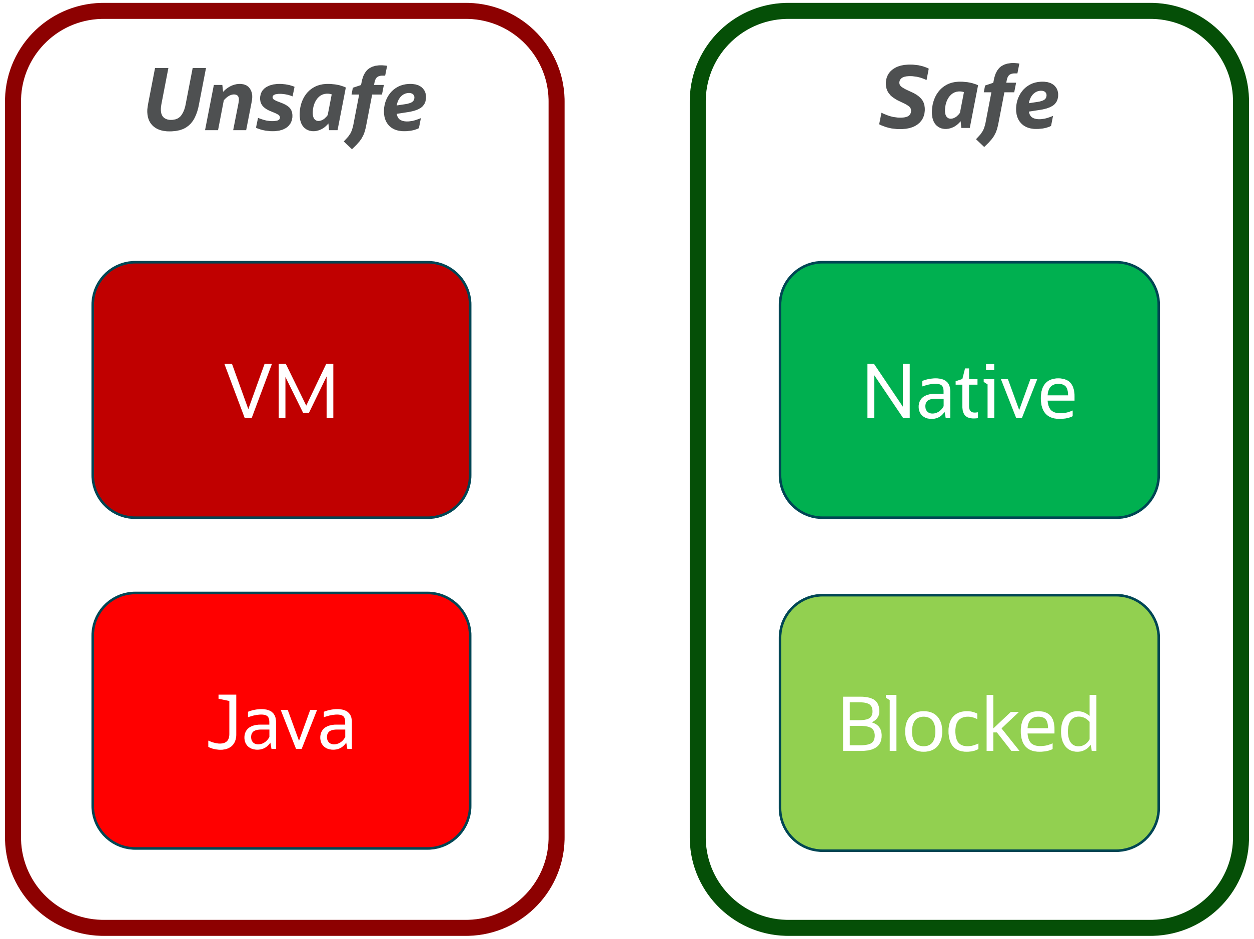
- Safepoints
- Handshakes
- Implementation
- Use cases
- Future use cases
- Additional functionality
- Real data

Safepoints

What is safe?

- Mutate virtual machine.
- Java thread (mutator) state.
- Common states:
  - Blocked
  - Native
  - Java
  - VM

Safepoints



Unsafe

VM

Java

Safe

Native

Blocked

Safepoints

States

- VM.
  - Transitions
- Java
  - Poll
  - Transitions
  - Elide into native.

- Native
  - Continue to execute
- Blocked
  - Stuck

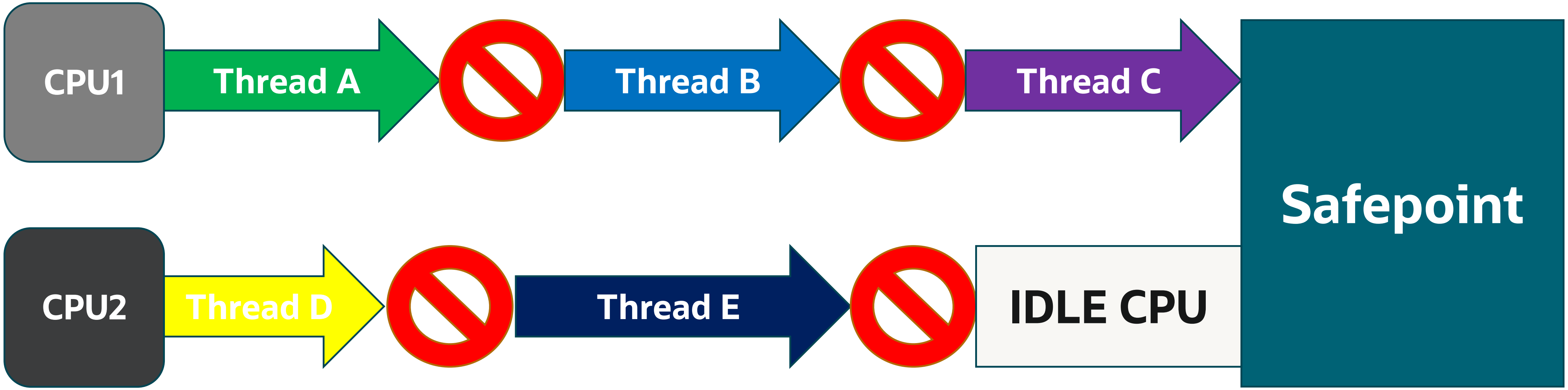
Safepoints

Basic execution

- VM Operation
- VM Queue.
- VM Thread, dispatch thread:
  - Stops
  - Accounts
  - Executes
  - Re-starts

Safepoints

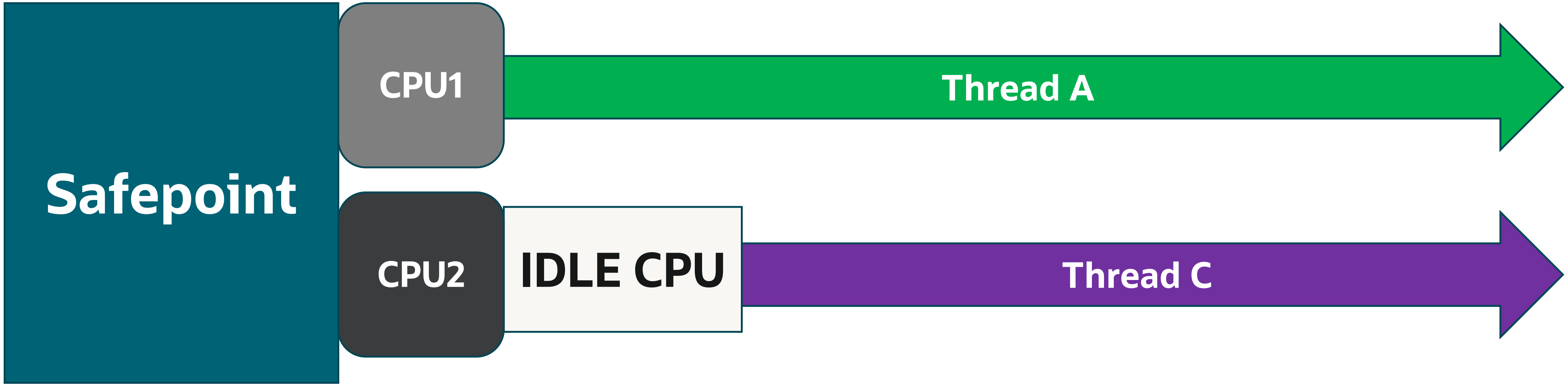
Synchronizing



🚫 = Poll

Safepoints

Restarting



Safepoints

Pros

- Simple model
- Global switch, constant time arming
- Single state
- Really safe
- Relative fast
  - ~200 us to reach a safepoint
  - ~200 us to reach full CPU utilization after safepoint

Safepoints

Cons

- Application intrusive
- Inter-thread dependency
- Unutilized CPU
- Operation without inter-thread dependency

Safepoints

Polling

```
// Generated poll in JIT
test rax, fixed-poll-adr

// Non trapping for non JIT code
bool SafepointMechanism::global_poll() {
       return (SafepointSynchronize::_state !=
               SafepointSynchronize::_not_synchronized);
}

// Arming
_state = _synchronizing;
mprotect(fixed-poll-adr, PAGE, MEM_PROT_NONE);

// Disarming
mprotect(fixed-poll-adr, PAGE, MEM_PROT_READ);
_state = _not_synchronized;
```

Handshakes

- Per thread safepoint
- Latency friendly
- Thread owned/local resources
  - Stack
  - Biased lock
  - Asynchronous exception
- Barrier
  - Un-publish
  - Handshake
  - Not visible

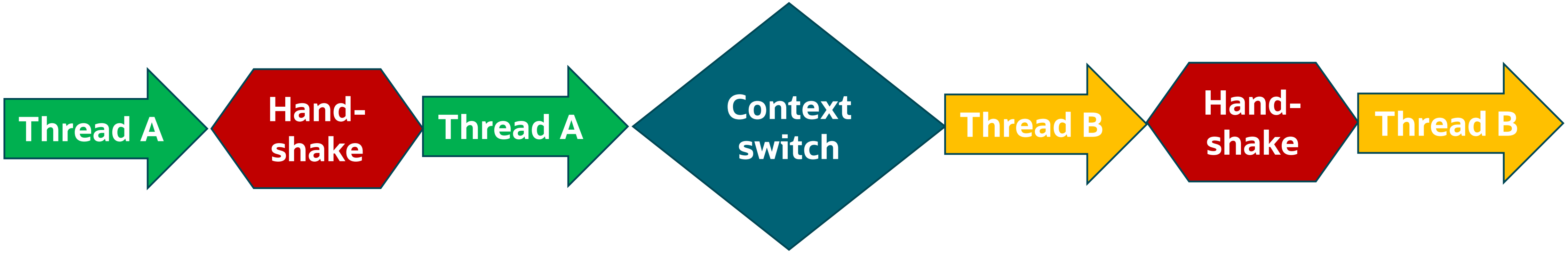
Handshakes

- VM Thread and queue
- VM Operation, non-safepoint
- Operation execution
  - Per thread installation of operation
    - Assign operation
    - Arms
  - Self (Java thread) processing
  - VM Thread processing

Handshakes

- Self processing
  - JIT poll
  - Transition
  - In slow path
    - Safepoint
    - Process handshake
- VM Thread processing
  - Safe Java threads
  - Stopped from entering unsafe state

Handshakes



- One CPU
- Serialized
- Time-slice

Handshakes

Pros

- ‘Unnoticeable’
- Individual threads
- Effectively barrier
  - Fence
- No dependencies
- CPU utilization

Handshakes

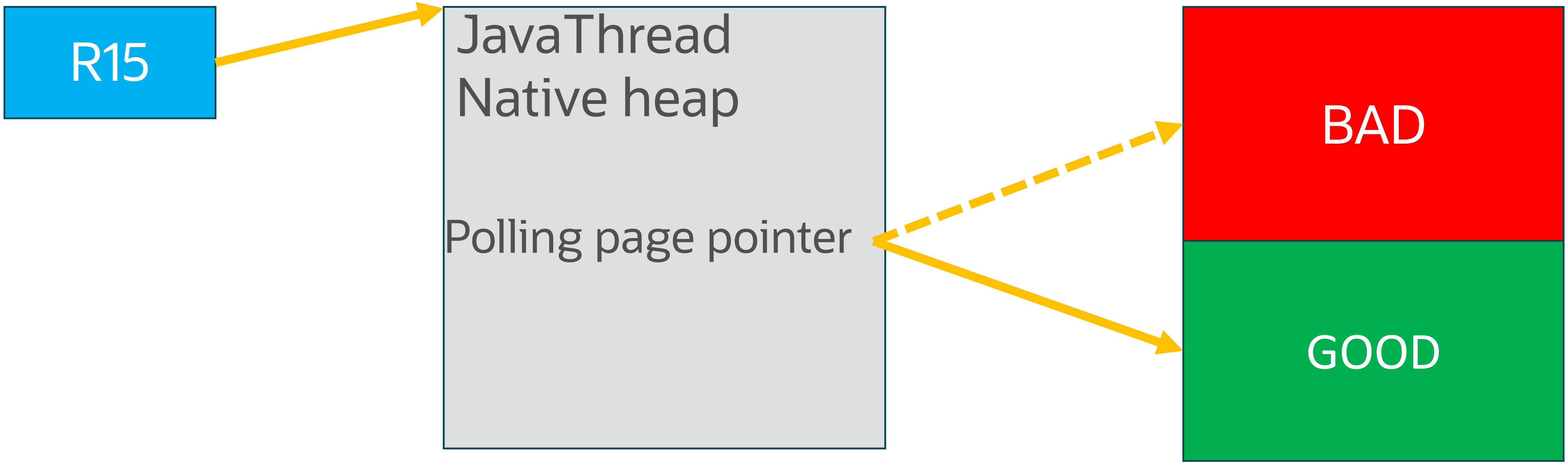
Cons

- Arming
- Complex states
- Limited use-cases
- Slow
  - ~20ms

Implementation

- Unique poll thread
- JavaThread* register
- Per thread polling page pointer
- Bad (unreadable) and good (readable)
- JIT, indirect load
- Non-JIT, branch-based

Implementation



R15

JavaThread Native heap

Polling page pointer

BAD

GOOD

Implementation

```
// Generated poll in JIT
mov  poll-offset + thread_reg, reg
test rax, reg

// Non trapping for non JIT code
bool SafepointMechanism::local_poll_armed(JavaThread* thread) {
  return thread->get_polling_word() & poll_bit();
}

// Arming one thread
thread->set_polling_page(poll_armed_value())

// Disarming one thread
thread->set_polling_page(poll_disarmed_value())

// Arming/disarming many threads
for (JavaThreadIteratorWithHandle jtiwh; JavaThread *cur = jtiwh.next(); ) {
  SafepointMechanism::arm_local_poll/disarm_local_poll(cur);
}
```

Use cases

- ZRendezvousClosure
- ShenandoahUnloadRendezvousClosure
- ZMarkFlushAndFreeStacksClosure
- InstallAsyncExceptionClosure
- RevokeOneBias
- DeoptimizeMarkedClosure
- NMethodMarkingClosure

Use cases

ZRendezvousClosure
ShenandoahUnloadRendezvousClosure

- Barrier handshake, no-op
- Stale metadata and nmethods synchronization
  - No polls while using metadata/nmethods
  1. Unpublish
  2. Handshake
  3. Free
- ZRendezvousClosure
  - Dead weak refs resurrection synchronization

Use cases

ZMarkFlushAndFreeStacksClosure

- Access thread local resource
- Flush per thread mark stacks
  - Available for concurrent GC work
- Free per thread stacks

Use cases

InstallAsyncExceptionClosure

- Access to thread local resource
- Asynchronous exception
- Thread.stop()
- ThreadDeath exception

Use cases

RevokeOneBias

- Thread owned resource
- Locked object
- Object points to a thread
- BasicLock -> Inflate -> Monitor
- Monitor handles contended locks

Use cases

DeoptimizeMarkedClosure

- Access to thread local resource
- Java stack
- nmethods marked for deoptimize
  - Classloading
  - Class redefinition
  - Invoke dynamic
- scan all stacks
- mark frames for deoptimize

Use cases

NMethodMarkingClosure

- Access to thread local resource
- Java stack
- Current nmethods on stack

Future use cases

- Suspend flag
- JVMTI
- Monitor deflation
- G1 StoreLoad barrier removal
- ZGC Concurrent stack scanning
- Safepoint via handshakes

Future use cases

Suspend flags

- Per thread
- Only checked in transition
- Overlapping functionality, but much less flexible
- Use-cases
  - Suspend/resume
  - Pending asynchronous exception (not same as installation)
  - Lazy critical native
  - JFR native sampling

Future use cases

JVMTI

- Lacking inter-thread dependencies
- Safepoint
- Suspend/resume
- Un-intrusive stack traces

Future use cases

Monitor deflation

- Contended object
- Monitor
- Address installed in markword
- Deflate
  - Remove monitor address from markword
- ABA, Monitor address resuse/free

Future use cases

G1 StoreLoad barrier removal

- StoreLoad in G1 post-write barrier
- Dirty cards
- Concurrent refinement, cleaning card(s)
- Handshaking a thread implies StoreLoad fence

Future use cases

ZGC Stack watermark barrier

- Concurrent stack scanning
- Require branch-based polling

Future use cases

Safepoints via handshakes

- Simplifies slow path
- Simplifies VM thread operation execution
- Preserve ordering

Additional functionality

- Direct handshakes, thread to thread
- Asynchronous handshakes
- Branch based polling

Additional functionality

Direct handshakes

- No VM thread hand-over
- Single handshake, no latency improvement
- Multiple handshake
  - Greatly parallelized
    - Throughput
    - Latency
  - A->B while C->D

Additional functionality

Asynchronous handshake

- Only target thread executed
- Per thread queue
  - Safepoint via handshake preserve ordering
- Suspend flag

Additional functionality

Branch based polling

- Selective polling
- Stack watermark barrier
  - Concurrent stack-scanning
- Asynchronous exception

Real data

A ZGC safepoint

```
|CPU0|CPU1|CPU2|CPU3|    Time
| J5 | ZD | J3 | J2 |    0 us
| J5 |*VT | J3 | J2 |   55 us
| J5 |*J1 | J3 | J2 |   86 us
| J5 |*.  | J3 | J2 |  110 us
| J5 | .  |*.  | J2 |  110 us
|*.  | .  | .  | J2 |  118 us
| .  | .  | .  |*.  |  129 us
| .  |*VT | .  | .  |  151 us
 SAFEPOINT, runtime/gc workers
| .  |  . |*VT | .  |  722 us
| .  |*J1 | VT | .  |  747 us
| .  |*J4 | VT | .  |  759 us
| .  |*J1 | VT | .  |  768 us
| .  | J1 | VT |*J2 |  802 us
| .  |*ZD | VT | J2 |  811 us
| .  | ZD |*J3 | J2 |  816 us
|*J5 | ZD | J3 | J2 |  827 us
```

JX = JavaThread X (green)
ZD = ZDriver (blue)
VT = VM Thread (red)

1. ZDriver initiates safepoint in CPU lane 1
2. VM Thread begins safepoint
3. VM Thread goes off-proc (for quicker stoppage)
4. Safepoint operation execution
5. VM Thread ends safepoint (starts the JavaThreads)
6. Notify ZDriver that the requested safepoint is completed

Real data

A ZGC handshake

```
|CPU0|CPU1|CPU2|CPU3|         Time
|*J2 | J4 | J1 | ZW |         0 us
|*VT | J4 | J1 | ZW |        11 us
| VT | J4 | J1 |*J3 |        14 us
|*J2 | J4 | J1 | J3 |     12970 us
|*VT | J4 | J1 | J3 |     25965 us
| VT | J4 | J1 |*ZW |     25995 us
|*J2 | J4 | J1 | ZW |     25998 us
```

JX = JavaThread X (green)
ZW = ZWorker (blue)
VT = VM Thread (red)

1. ZWorker initiates a handshake in CPU lane 3
2. VM Thread starts executing the handshake
3. Notify ZWorker, handshake is completed

Thank You

Runtime

Mutexes and TSM

- Type-Stable-Memory
- Mutexes

Runtime

Java Threads

- Linked list
- Mutex
- Array
- Hazard pointers

Runtime

Hash tables

- Fixed size
- Mutex
- Spinlock
- TSM
- Concurrent
- EBR/RCU
- Constant-time reads

Runtime

Fast safepoints

- Mutex backed
- Serialized
  - Stopping
  - Wake-up
- Futex
- Co-op semaphore