# Loci: Thread-Locality for Java-Like Languages


Tobias Wrigstad

tobias.wrigstad@it.uu.se


alumni

*Filip Pizlo* *Fadi Meawad* *Lei Zhao* *Jan Vitek*

# Outline

- Motivating thread-locality
- An overview of support for thread-locality in mainstream languages
- Approaches to typing thread-locality
- Consequences for logical memory layout
- Formal validation
- Implementation & practical validation
- Extending Loci to full Java
- Shortcomings
- Future work: Loci 2.0

# Thread-Local Data

- Easy to reason about

  No races

  No deadlocks

  Worst-case execution time-analysis (e.g., thin locks, biased locks)

- Enable optimisations

  Simplifies parallel garbage collection

  Lock eliding

  Cache synchronisation

- **NB** Thread-Local ≠ Stack-Local

  Semantic property of origin of accesses, not storage

# Thread-Locality Support in Mainstream JLLs

- Studied: Java, C#, various C++ impls, Object Pascal, Python and Ruby

  Implemented through library-support (possibly with annotations to boot)

  Focus on dealing with thread-local contents of **global** fields

- No guarantees unless you use the library consistently

  Regularly comes with performance penalties (*e.g.,* hash-map indirection)

  Penalties mean approach does not scale to whole-program use

- In conclusion

  – No static guarantees (no or few optimisations possible)

  – Bad traceability (no real reasoning aid)

  \+ Do not complicate language semantics

# Example: Java's ThreadLocal API (logically)

```
class ThreadLocal<T> {
  Map<Thread,T> data = new HashMap<Thread,T>();

  void set(T obj) {
    Thread current = Thread.currentThread();
    data.put(current, obj);
  }

  T get() {
    Thread current = Thread.currentThread();
    return data.get(current);
  }
}
```

# Running Example of Thread-Local Data Use

```
class Example {
  private Foo foo = null; // Should be a thread-local value
  private Foo bar = null; // Possibly shared value

  void frob() {
    bar = foo; // Leak!
  }
}
```

# Using java.lang.ThreadLocal

```
class Example {
  private ThreadLocal<Foo> foo = new ThreadLocal<Foo>();
  private Foo bar = null; // Possibly shared value

  void frob() {
    bar = foo.get(); // Reading foo requires indirection
  }
}
```

# Using java.lang.ThreadLocal

```
class Example {
  private ThreadLocal<Foo> foo = new ThreadLocal<Foo>();
  private Foo bar = null; // Possibly shared value

  void frob() {
    bar = foo.get(); // Leak!
  }
}
```

# Our Goal (as Given to us by Doug Lea)

- ”Create a simple type system for thread-local data that we can use in Java 7”

- Functional requirements

  Statically guarantee thread-locality to enable reasoning and optimisations

- Non-functional requirements

  No alteration of language semantics

  Easy enough to be used by average, mainstream programmers

  Work with existing Java legacy code

# Hypothesis

- Most instances of a single class have the same thread-locality

  Either mostly thread-local;

  Or mostly shared between threads

# Straw-man Proposal

- Divide classes into two categories

  Classes whose instances are always thread-local

  Classes whose instances are always shared between threads

- Analysis

  \+ Easy to understand

  \+ Easily traceable (type gives thread-locality)

  \+ No modification of existing language semantics needed

  – Unreasonable requirement for library classes

# Hypothesis, revisited

- Most instances of **application-specific** classes have the same thread-locality

  Either mostly thread-local;

  Or mostly shared between threads

- Very informal inspection of Java code ”verifies” assumption

  Proper scientific studies are needed

  I strongly believe we have found a reasonable design trade-off

# Revised Proposal

- Divide classes into two categories

  Classes whose instances are safe for use as thread-local data

  Classes whose instances are not safe for use as thread-local data

- A class is safe to use to create thread-local instances if

  It never stores a reference to itself in a location that could be accessed by another thread

  *i.e.,* a global field, or a field of a shared object

- Use type annotations to track thread-locality by **use**, not declaration

- Implement thread-local fields using Java's ThreadLocal API

# Revised Proposal (cont'd)

- Analysis
  - + Easy to understand (dataflow must preserve thread-locality)
  - + Easily traceable (type and/or annotation gives thread-locality)
  - + No modification of existing language semantics needed
  - + Delayed thread-locality decision works with library classes
  - – Additional syntactic baggage

# Loci

- Two annotations for types (and classes)

  `@Thread` and `@Shared`

  A `@Shared` class always creates shared values

  A `@Shared` variable (field, etc.) points to a shared value

  A `@Thread` class **can be used** to create thread-local values

  A `@Thread` variable (field, etc.) points to a thread-local value

- Example of dataflow constraint

  Cannot store instances of `@Shared` classes in `@Thread` typed variables

- Suitable for Java-like languages, pluggable type system (with twist)

# Demo

```
@Shared class Example {
  private @Thread Foo foo = null;
  private @Shared Foo bar = null;

  void frob() {
    bar = foo; // Will not compile
  }
}
```

# Demo

Example always creates shared instances

```
@Shared class Example {
  private @Thread Foo foo = null;
  private @Shared Foo bar = null;

  void frob() {
    bar = foo; // Will not compile
  }
}
```

holds thread-local value

holds shared value

breaks dataflow constraint

# Reducing Syntactic Overhead

- Classes default to `@Shared`

  This gives a valid Loci semantics to legacy Java code

- Types default to ”same as enclosing instance”

  **NB** instance ≠ class

- No need to use Java’s ThreadLocal API explicitly

  `@Thread T f` *expands to*

```
@Shared ThreadLocal<T> f = new ThreadLocal<T>().
```

- Result: very few annotations needed in practise (++)

# Demo of Defaults (last code example)

```
@Shared class Example {
  private @Thread Foo foo = null;
  private @Shared Foo bar = null;

  void frob() {
    bar = foo;
  }
}
```

# Demo of Defaults (expand `@Thread` annotations)

```
@Shared class Example {
  private @Shared ThreadLocal<Foo> foo = new ThreadLo...;
  private @Shared Foo bar = null;

  void frob() {
    bar = foo.get();
  }
}
```

# Demo of Defaults (remove explicit defaults)

```
class Example {
  private ThreadLocal<Foo> foo = new ThreadLocal<Foo>();
  private Foo bar = null;

  void frob() {
    bar = foo.get();
  }
}
```

# Full Circle (we have returned to original code), Big Difference

```
class Example {
  private ThreadLocal<Foo> foo = new ThreadLocal<Foo>();
  private Foo bar = null;

  void frob() {
    bar = foo.get(); // Will not compile
  }
}
```

# Penalties of using Java's ThreadLocal API

- Micro benchmarks suggest hashmap indirection is 8x time of field access

  Plus additional overhead of creating, GC'ing etc. hashmap objects

- Hashmaps are expensive, multiple hashmaps per object is unreasonable

  Likely to be a factor 30 overhead for class with many thread-local fields

- Most hashmaps would have single element

  Because a thread-local *field* of a thread-local *object* will never be accessed by more than a single thread

  **Thus, for such fields, we need no expansion before compiling**

# Demo

No expansion needed for either field

```
@Thread class Foo {
  Foo foo = null; // Thread-local if this is
  Foo bar = null; // Thread-local if this is

  void frob() {
    bar = foo; // OK, same thread-locality
  }
}

@Thread Foo x = new Foo();
@Shared Foo y = new Foo();
x.foo = y.foo; // Will not compile
```

# Interplay with Inheritance

- Subclassing must be preserve annotations

  Alternative solutions possible, but are complex and seemingly of little use

- Common root class is a problem

  What is the annotation on `java.lang.Object`?

  **Pragmatic solution:** make exception to rule above for Object

  We are now free to annotate Object `@Thread`

# Static Guarantees & Consequences

- Thread-locality of an expression's return value is immediately obtained from its type

  A conservative approximation, `@Shared` type $\not\Rightarrow$ actually shared value

  `@Thread` type $\Rightarrow$ thread-local value

- Memory **logically** gets partitioned into **thread-local** heaps, plus **shared** heap

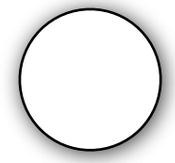
  We can *e.g.* garbage collect thread-local heaps in parallel

  Simplify cache coherence in compilers (?)

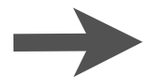
  Lock-taking on `@Thread` values can be removed at compile-time

# Memory-partitioning in Loci (logical)

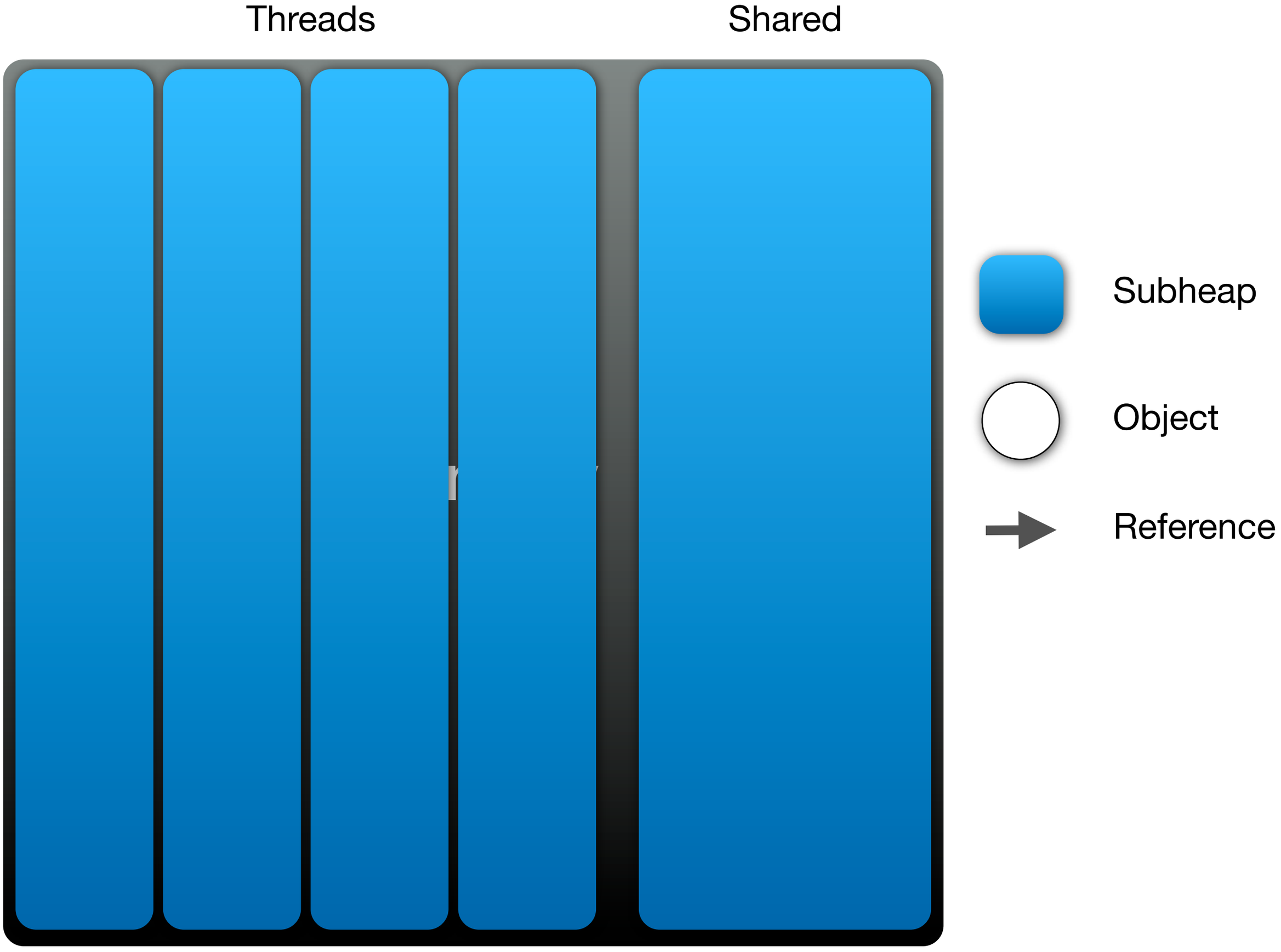
Subheap



Object

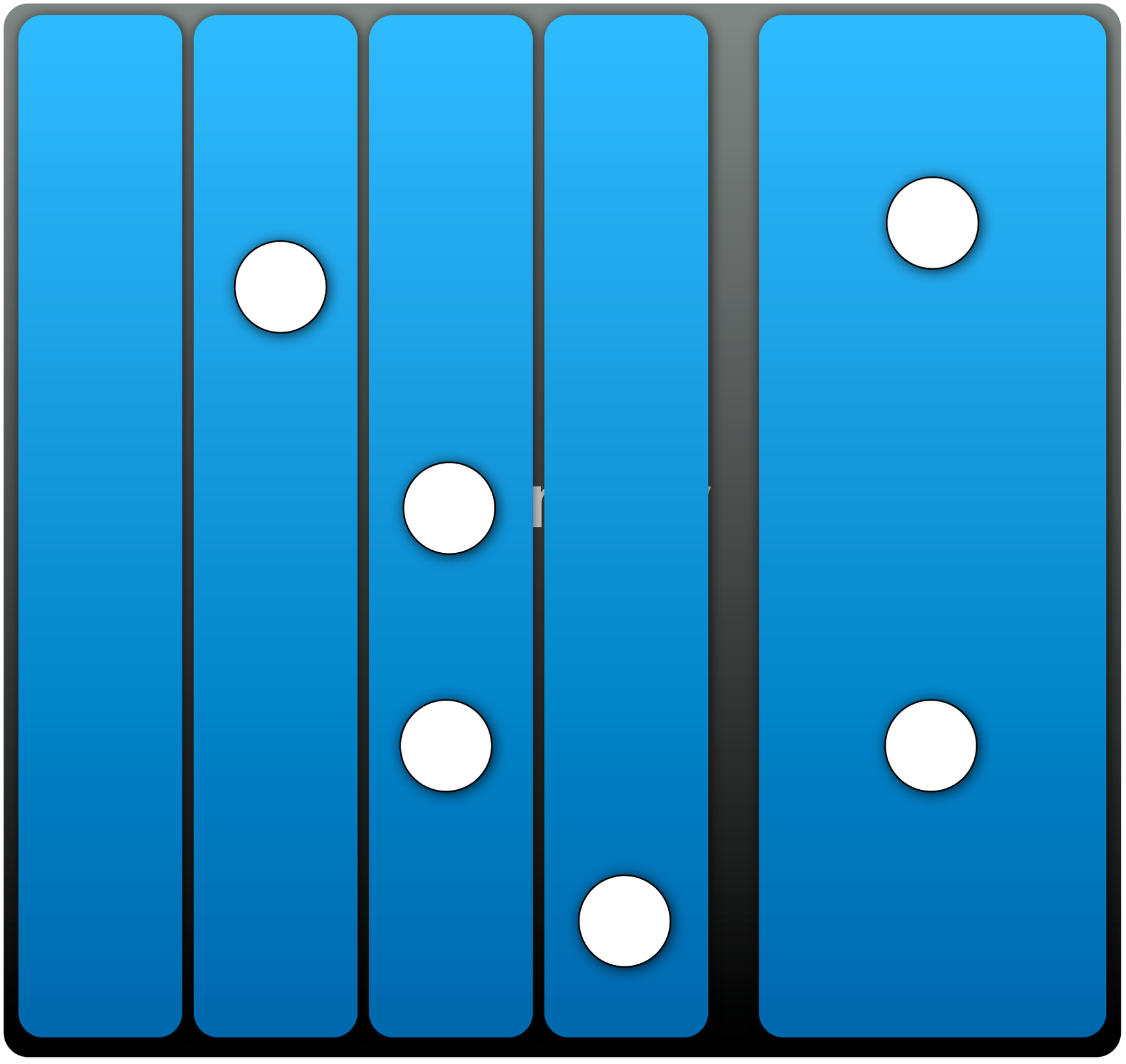


Reference


Threads
Shared
Subheap
Object
Reference

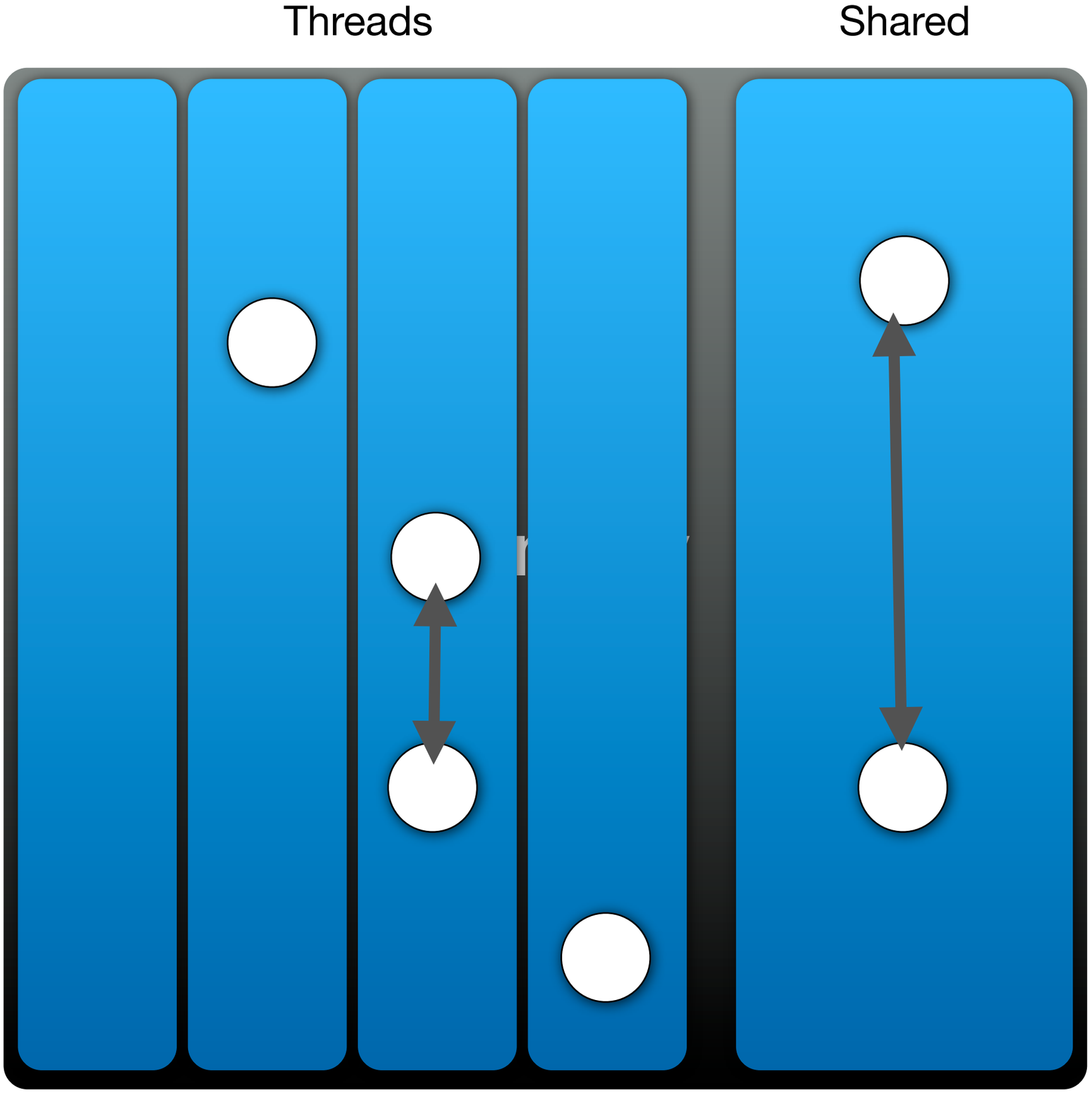
Threads

Shared



**Allowed**

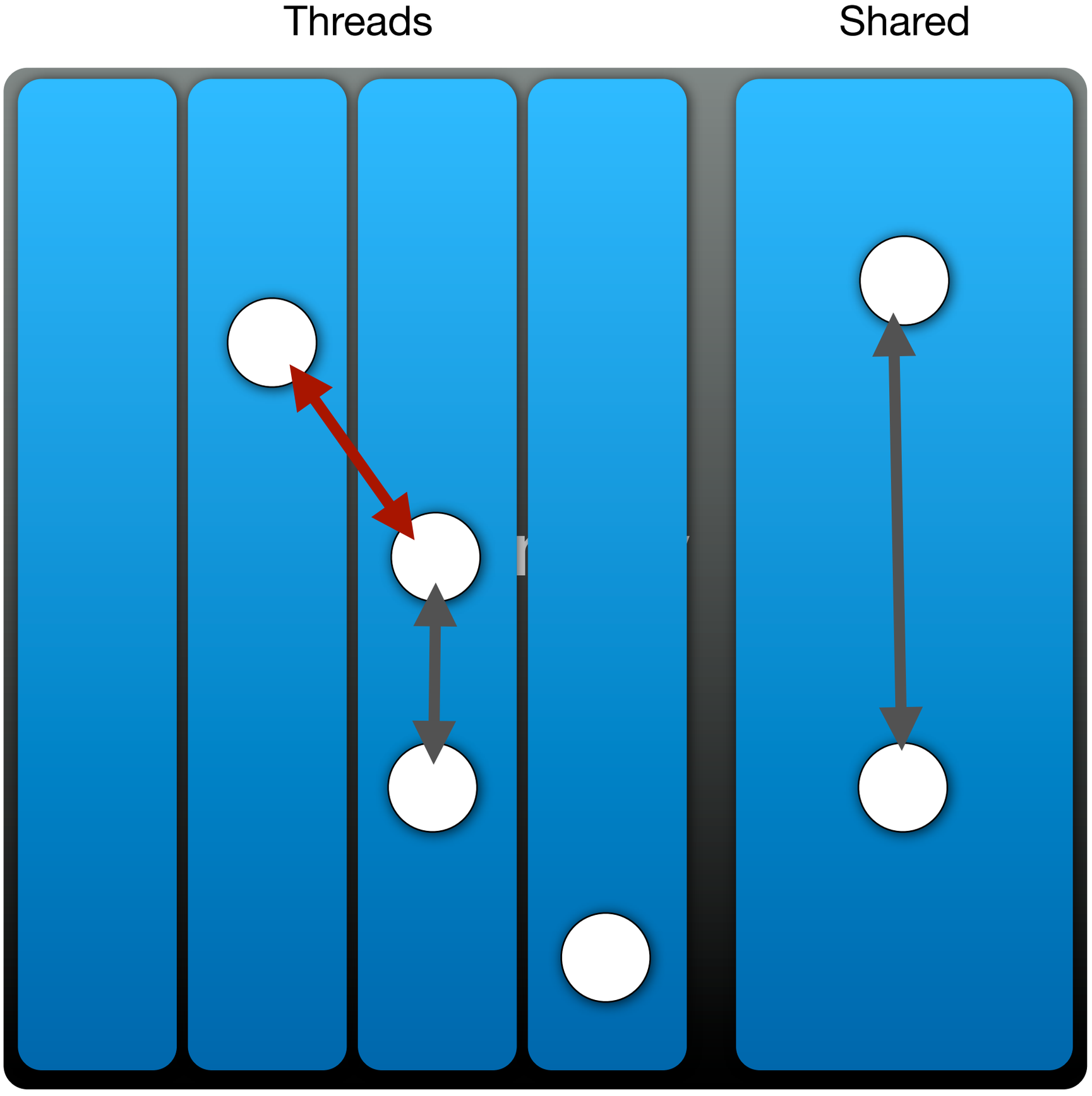
**Disallowed**




**Allowed**

*Intra-thread & intra-shared*

**Disallowed**

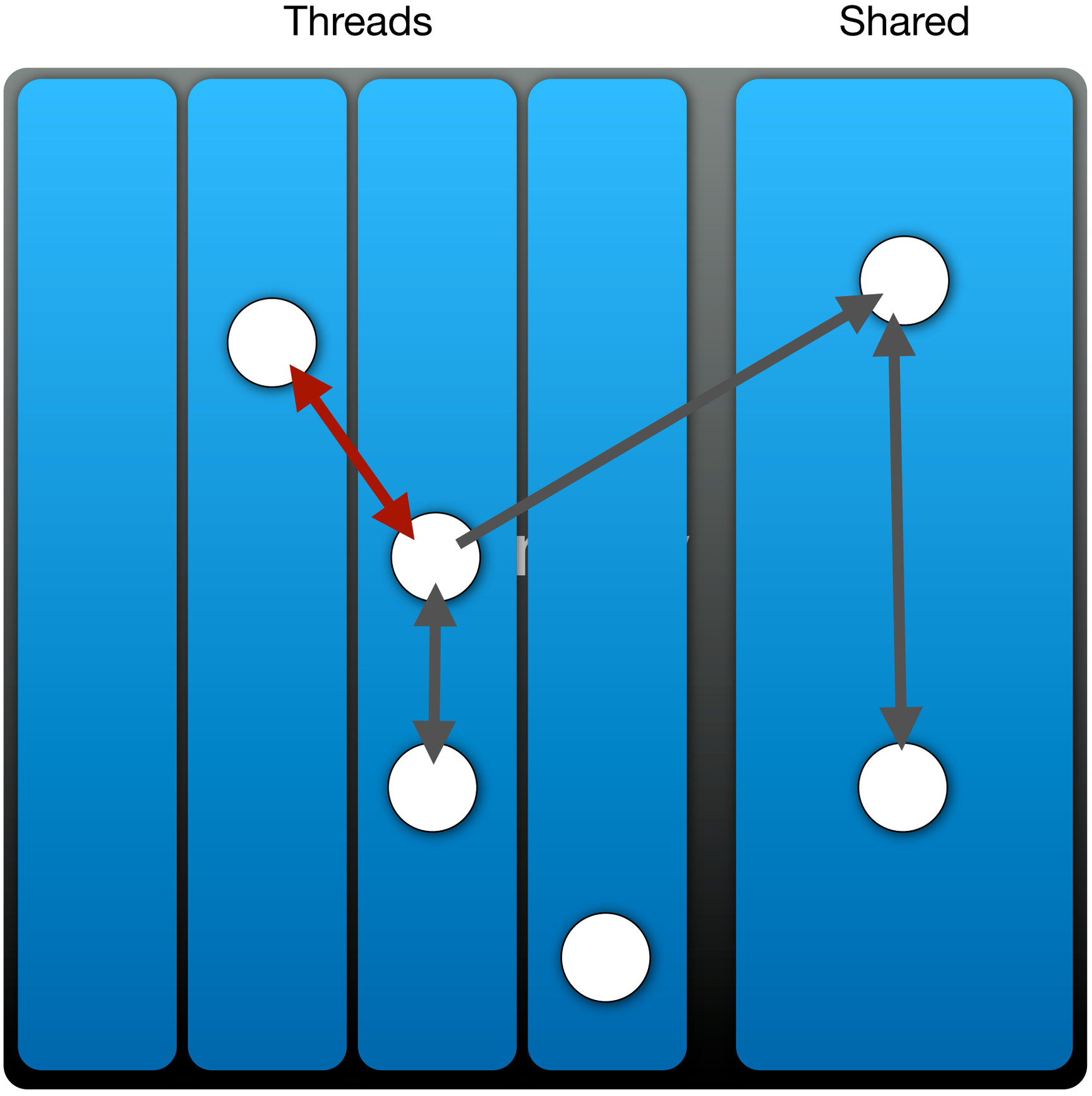



**Allowed**

Intra-thread & intra-shared

**Disallowed**

*Inter-thread*



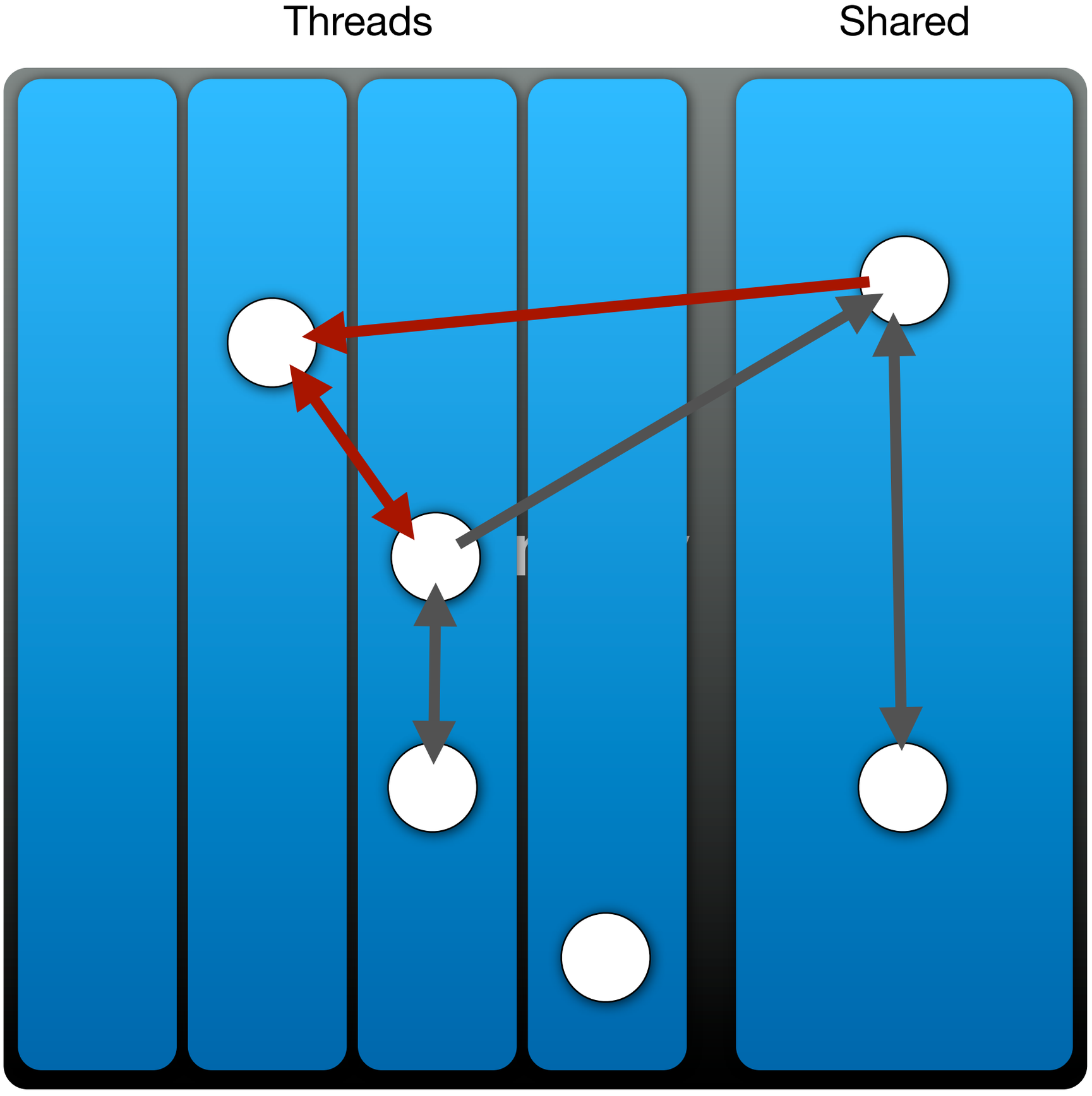

**Allowed**

Intra-thread & intra-shared

*Thread to shared*

**Disallowed**

Inter-thread




**Allowed**

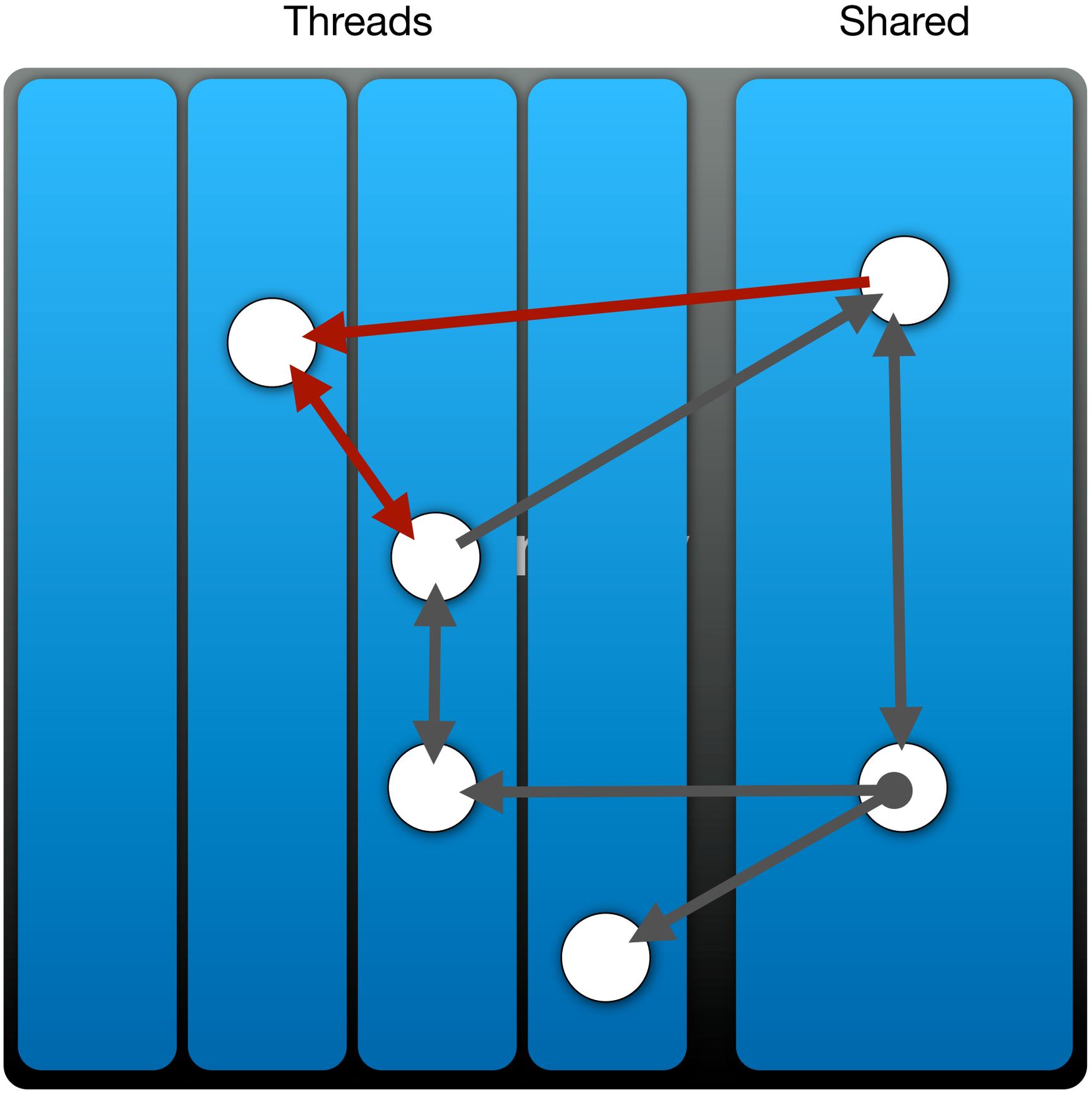
Intra-thread & intra-shared

Thread to shared

**Disallowed**

Inter-thread

*Shared to thread*




**Allowed**

Intra-thread & intra-shared

Thread to shared

**Disallowed**

Inter-thread

Shared to thread
*unless guarded by thread-local fields*

# Loci Formalisation (1)

- Formalised as a core Java-like language with standard simplifications
  - No interfaces
  - No implicit subsumption
- Model thread-locality as an ownership types system with threads-as-owners
  - Simple, well-known technique
  - Default annotation on types can be modelled by `owner`
  - Shared heap can be modelled by `world`
- Standard type soundness theorem (progress + preservation)
- Variable and field access theorems for proving thread-locality

# Loci Formalisation (2)

*Loci Syntax*

$$
\begin{array}{rcll}
P & ::= & \overline{cd} & program \\
cd & ::= & \alpha\ \texttt{class}\ c\ \texttt{extends}\ d\ \{\ \overline{fd}\ \overline{md}\ \} & class\ declaration \\
fd & ::= & \tau\, f & field \\
md & ::= & \tau\, m(\overline{\tau}\ \overline{x})\ \{\ s\texttt{;return}\ y\ \} & method \\
s & ::= & s\texttt{;}s \mid \texttt{skip} \mid x = y.f \mid x = y \mid y.f = z \mid \tau\ x \mid & statement \\
 & & x = \texttt{new}\ \tau() \mid x = y.m(\overline{z}) \mid x = \texttt{start}\ c() & \\
\tau & ::= & \alpha\ c & type \\
\alpha & ::= & \texttt{@Thread} \mid \texttt{@Shared} \mid \texttt{@Context} & annotations \\
 & & & \\
E & ::= & [] \mid E[x : \tau] & local\ type\ environment
\end{array}
$$

Context-dependent default made explicit in type system

# Viewpoint Adaptation (Standard Ownership Types Trick)

- Remember: Type of a field lookup (etc.) depends on thread-locality of target

```
@Thread class Foo { @Context Bar f; }

@Thread Foo x;
@Shared Foo y;

x.f // @Thread Bar          @Thread ⊕ @Context = @Thread
y.f // @Shared Bar          @Shared ⊕ @Context = @Shared
```

viewpoint adaptation operator used in the type system

$$\alpha_1 \oplus \alpha_2\ c = \begin{cases} \alpha_1\ c \text{ if } \alpha_2 = \texttt{@Context} \\ \alpha_2\ c \text{ otherwise} \end{cases}$$

$$\alpha_1 \oplus \alpha_2 \oplus \alpha_3\ c = \alpha_1 \oplus (\alpha_2 \oplus \alpha_3\ c)$$

# Loci Formalisation (3)

Subtyping preserves annotations

$$\frac{\vdash \alpha\ c \qquad \vdash \alpha\ d \qquad c \leq d}{\alpha\, c \leq \alpha\, d}\ (\textsc{sub-annote})$$

$$\frac{E(y) = \alpha\ c \quad \mathsf{fields}(c.f) = \tau' \quad \alpha \oplus \tau' \leq E(x)}{E \vdash x = y.f; E}\ (\textsc{t-select})$$

$$\frac{E(y) = \alpha\ c \quad \mathsf{fields}(c.f) = \tau \quad E(z) \leq \alpha \oplus \tau}{E \vdash y.f = z; E}\ (\textsc{t-update})$$

$$\frac{\vdash \tau \qquad \tau \leq E(x)}{E \vdash x = \mathbf{new}\ \tau(); E}\ (\textsc{t-new})$$

$$\frac{E(y) = \alpha\ c \quad \mathsf{mtype}(c.m) = \overline{\tau} \to \tau' \quad E(\overline{z}) \leq \alpha \oplus \overline{\tau} \quad \alpha \oplus \tau' \leq E(x)}{E \vdash x = y.m(\overline{z}); E}\ (\textsc{t-call})$$

$$\frac{\mathsf{mtype}(c.\mathtt{run}) = \epsilon \to \tau \quad E(x) = \mathtt{@Shared}\ c}{E \vdash x = \mathbf{start}\ c(); E}\ (\textsc{t-fork})$$

# Loci Formalisation (3)

$$\frac{\vdash \alpha\ c \qquad \vdash \alpha\ d \qquad c \leq d}{\alpha\, c \leq \alpha\, d}\ (\textsc{sub-annote})$$

$$\frac{E(y) = \alpha\ c \quad \mathsf{fields}(c.f) = \tau' \quad \alpha \oplus \tau' \leq E(x)}{E \vdash x = y.f; E}\ (\textsc{t-select})$$

$$\frac{E(y) = \alpha\ c \quad \mathsf{fields}(c.f) = \tau \quad E(z) \leq \alpha \oplus \tau}{E \vdash y.f = z; E}\ (\textsc{t-update})$$

Viewpoint adaptation applied for correct field typing

$$\frac{\vdash \tau \qquad \tau \leq E(x)}{E \vdash x = \mathbf{new}\ \tau(); E}\ (\textsc{t-new})$$

$$\frac{E(y) = \alpha\ c \quad \mathsf{mtype}(c.m) = \overline{\tau} \to \tau' \quad E(\overline{z}) \leq \alpha \oplus \overline{\tau} \quad \alpha \oplus \tau' \leq E(x)}{E \vdash x = y.m(\overline{z}); E}\ (\textsc{t-call})$$

$$\frac{\mathsf{mtype}(c.\mathtt{run}) = \epsilon \to \tau \quad E(x) = \mathtt{@Shared}\ c}{E \vdash x = \mathbf{start}\ c(); E}\ (\textsc{t-fork})$$

# Loci Formalisation (3)

$$\frac{\vdash \alpha\ c \qquad \vdash \alpha\ d \qquad c \leq d}{\alpha\, c \leq \alpha\, d}\ (\textsc{sub-annote})$$

$$\frac{E(y) = \alpha\ c \quad \mathsf{fields}(c.f) = \tau' \quad \alpha \oplus \tau' \leq E(x)}{E \vdash x = y.f; E}\ (\textsc{t-select})$$

$$\frac{E(y) = \alpha\ c \quad \mathsf{fields}(c.f) = \tau \quad E(z) \leq \alpha \oplus \tau}{E \vdash y.f = z; E}\ (\textsc{t-update})$$

$$\frac{\vdash \tau \qquad \tau \leq E(x)}{E \vdash x = \mathbf{new}\ \tau(); E}\ (\textsc{t-new})$$

$$\frac{E(y) = \alpha\ c \quad \mathsf{mtype}(c.m) = \overline{\tau} \to \tau' \quad E(\overline{z}) \leq \alpha \oplus \overline{\tau} \quad \alpha \oplus \tau' \leq E(x)}{E \vdash x = y.m(\overline{z}); E}\ (\textsc{t-call})$$

$$\frac{\mathsf{mtype}(c.\mathtt{run}) = \epsilon \to \tau \quad E(x) = \mathtt{@Shared}\ c}{E \vdash x = \mathbf{start}\ c(); E}\ (\textsc{t-fork})$$

Viewpoint adaptation applied for method typing

# Loci Formalisation (3)

$$\frac{\vdash \alpha\ c \qquad \vdash \alpha\ d \qquad c \leq d}{\alpha\, c \leq \alpha\, d}\ (\textsc{sub-annote})$$

$$\frac{E(y) = \alpha\ c \quad \mathsf{fields}(c.f) = \tau' \quad \alpha \oplus \tau' \leq E(x)}{E \vdash x = y.f; E}\ (\textsc{t-select})$$

$$\frac{E(y) = \alpha\ c \quad \mathsf{fields}(c.f) = \tau \quad E(z) \leq \alpha \oplus \tau}{E \vdash y.f = z; E}\ (\textsc{t-update})$$

$$\frac{\vdash \tau \qquad \tau \leq E(x)}{E \vdash x = \mathbf{new}\ \tau(); E}\ (\textsc{t-new})$$

$$\frac{E(y) = \alpha\ c \quad \mathsf{mtype}(c.m) = \overline{\tau} \to \tau' \quad E(\overline{z}) \leq \alpha \oplus \overline{\tau} \quad \alpha \oplus \tau' \leq E(x)}{E \vdash x = y.m(\overline{z}); E}\ (\textsc{t-call})$$

$$\frac{\mathsf{mtype}(c.\mathbf{run}) = \epsilon \to \tau \quad E(x) = \mathtt{@Shared}\ c}{E \vdash x = \mathbf{start}\ c(); E}\ (\textsc{t-fork})$$

**NB** all Threads are `@Shared`

# Loci Formalisation (4)

**Theorem 1.** *Local variables point into shared heap or current heaplet. If* $\Gamma; E \vdash H; \overline{T}\,(S\,\langle F, s\rangle, \rho)$, *then* $\forall \iota \in rng(F).\ \mathsf{tid}(H, \iota) \in \{\varrho, \rho\}$.

✓

Heap

Top-most stack-frame

Id of current thread

Id of shared heap

**Theorem 2.** *Field accesses yield pointers to shared heap or current heaplet. Let* $s$ *be a field access* $x = y.f$. *If* $\Gamma; E \vdash H; \overline{T}\,(S\,\langle F, s\rangle, \rho)$, $H; \overline{T}\,(S\,\langle F, s\rangle, \rho) \rightarrow H'; \overline{T'}\,(S'\,\langle F', s'\rangle, \rho)$, *and* $F'(x) = \iota$, *then* $\mathsf{tid}(H', \iota) \in \{\varrho, \rho\}$.

✓

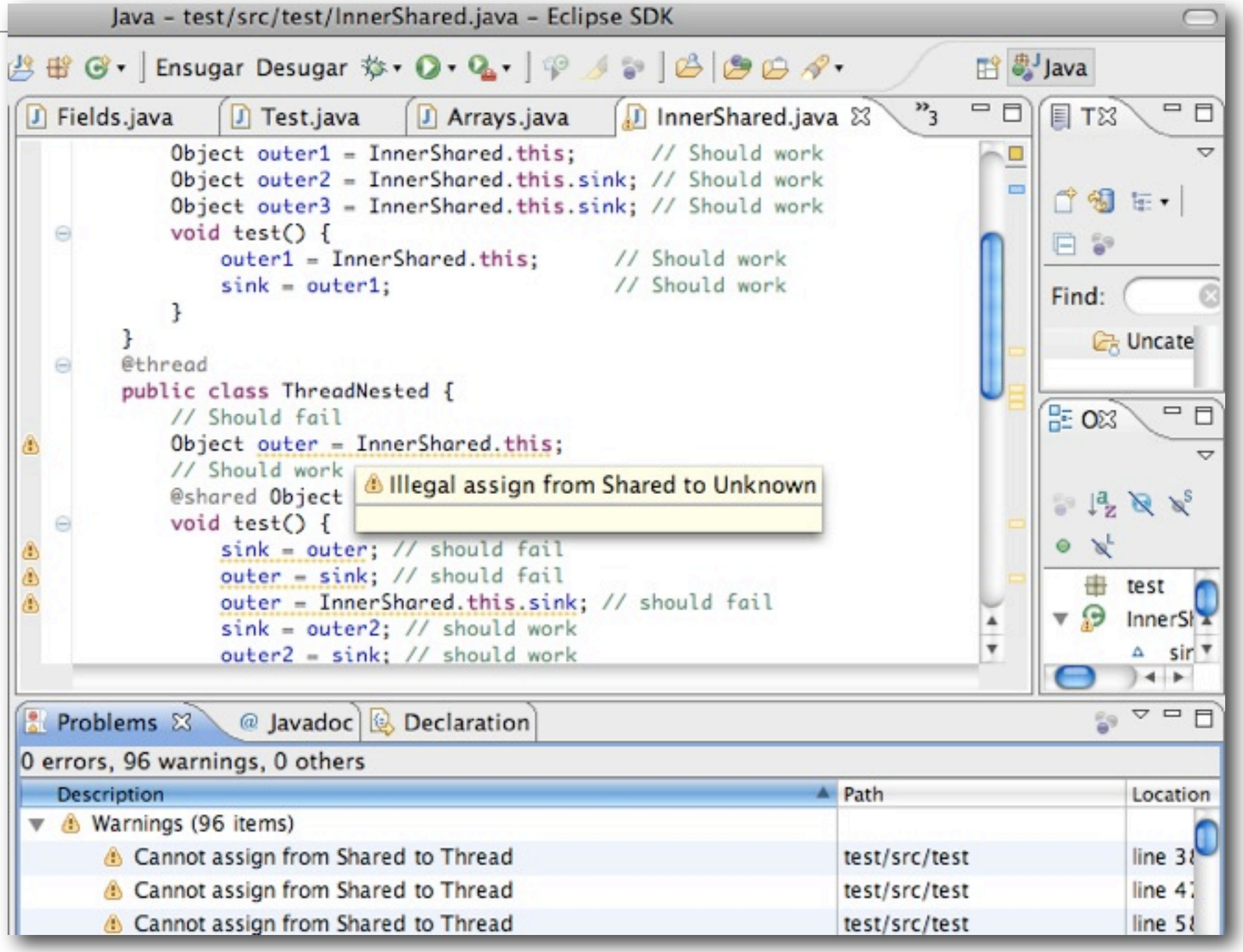
| | | | |
|---|---|---|---|
| $H ::= [] \mid H[\iota \mapsto o]$ | *heap* | $F ::= [] \mid F[y \mapsto v]$ | *stack frame* |
| $T ::= (S, \rho) \mid (\mathtt{NPE}, \rho)$ | *thread* | $o ::= c(\rho, F)$ | *object* |
| $S ::= \epsilon \mid S\,\langle F, s\rangle$ | *stack* | $v ::= \iota \mid \mathtt{null}$ | *value* |

# Loci Implementation

- Prototype implementation as an Eclipse Plugin

  Does not handle all aspects of Java well

  Subsequently generates warnings rather than errors




*Can give demo later*

# Practical Validation (1)

| Inferred | Classes |
|---|---|
| `@Thread` | 5996 |
| `@Shared` | 1289 |
| Σ | **7285** |

Most classes in GNU classpath could be annotated `@Thread`

A lot of thread-locality in actual programs

| **Average Thread-Locality Rate** (%) | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | **ANTLR** | **BLOAT** | **Eclipse** | **Apache FOP** | **HSQLDB** | **Jython** | **Lucene Index** | **Lucene Search** | **PMD** | **Xalan** |
| **Objects** | 79 | 82 | 63 | 78 | 88 | 77 | 78 | 74 | 83 | 66 |
| **Bytes** | 71 | 77 | 64 | 76 | 85 | 73 | 71 | 51 | 81 | 69 |

[Analysis and inferencer by Filip Pizlo]

# Practical Validation (2)

Added Loci annotations to ~44 KLOC Java

| | **LOC** | **Classes** | `@Thread` | `@Shared` | **Default** |
|---|---|---|---|---|---|
| **Raytracer** | 1496 | 16 | 16 | 0 | 0 |
| **Lucene Search** | 42749 | 285 | 19 | 0 | 266 |
| Σ | 44245 | 301 | 35 | 0 | 266 |

| | **# `@Thread` Annotations** | | | | **# `@Shared` Annotations** | | | |
|---|---|---|---|---|---|---|---|---|
| | **Fields** | **Params** | **Returns** | **Vars** | **Fields** | **Params** | **Returns** | **Vars** |
| **Raytracer** | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| **Lucene Search** | 2 | 0 | 4 | 5 | 0 | 20 | 0 | 9 |
| Σ | 2 | 0 | 4 | 6 | 1 | 21 | 1 | 10 |

[Case studies by Fadi Meawad and Lei Zhao]

# Practical Validation (3)

- Loci seems to be compatible with how people write Java programs
- There is a lot of thread-locality out there (unsurprisingly)
- Not having an annotated Java API renders much worse results

  Not feasible in practise

  Good news is most Java API classes can be made `@Thread`

  **NB** Including utility classes such as Lists, Maps and Sets

- Future work!

# Extending Loci to Full Java

**NB** Not formally proven correct, merely conjectured

- ✓ Anonymous classes
- ✓ Arrays
- ✓ Constructors
- ✓ Exceptions
- ✓ Inner and nested classes
- ✓ Interfaces
- ✓ Packages
- ✓ Statics

- Work in progress:

  Generics

  Reflection does not work with `@Thread` data (insolvable?)

# Extending Loci to Full Java

**NB** Not formally proven correct, merely conjectured

- ✓ Anonymous classes **STRAIGHTFORWARD**
- ✓ Arrays *(relatively tricky)*
- ✓ Constructors **SAFE BY CONSTRUCTION**
- ✓ Exceptions **ALWAYS THREAD-LOCAL**
- ✓ Inner and nested classes **STRAIGHTFORWARD, SOME RESTRICTIONS**
- ✓ Interfaces *(a little tricky)*
- ✓ Packages **CAN BE USED TO CONTROL DEFAULT ANNOTATIONS**
- ✓ Statics **DEFAULT TO** `@Shared`

- Work in progress:

  Generics **CAN BE USED TO IMPROVE STATIC METHODS**

  Reflection does not work with `@Thread` data (insolvable?)

# Extending Loci to Full Java

**NB** Not formally proven correct, merely conjectured

- ✓ Primitive type
- ✓ Immutable objects
- ✓ ThreadLocal API
- ✓ Thread

- Work in progress:

  Immutables declared by programmer

# Extending Loci to Full Java

**NB** Not formally proven correct, merely conjectured

- ✓ Primitive type **SAFE TO IGNORE DUE TO VALUE SEMANTICS**
- ✓ Immutable objects **SAFE TO IGNORE FOR BUILT-IN TYPES**
- ✓ ThreadLocal API **CANNOT BE ANNOTATED, ASSUMED TO BE CORRECT**
- ✓ Thread **DEFAULT TO** `@Shared`

- Work in progress:

  Immutables declared by programmer **HARD TO CHECK, PRAGMATIC SOLUTION SIMPLY IGNORES CLASSES MARKED IMMUTABLE**

# Example: Thread-Locality and Java Arrays (1)

- Common root classes pose problems for typing arrays (and in general)

  How express thread-locality for both the array **and** its elements?

  Naïve solution: use two annotations:

| Array | Elements | Interpretation |
|---|---|---|
| `@Thread` | `@Thread` | Thread-local array of thread-local objects |
| `@Thread` | `@Shared` | Thread-local array of shared objects |
| `@Shared` | `@Shared` | Shared array of shared objects |
| `@Shared` | `@Thread` | Shared array of thread-local objects |

# Example: Thread-Locality and Java Arrays (2)

- Cannot use two annotations for arrays due to subsumption

```
@Shared @Thread Object[] x;
@Shared Object y = x; // Legal, x is an object
(@Shared @??? Object[]) y; // How recover element Thread-
                                                    Locality?
```

- Three solutions (at least)
  - Use special annotations for arrays
  - Store objects' thread-locality at run-time
  - Require elements to have same thread-locality as the array

- Similar issues arise with parametrically polymorphic classes too

# Example: Thread-Locality and Java Arrays (3)

- Cannot use two annotations for arrays due to subsumption

```
@Shared @Thread Object[] x;
@Shared Object y = x; // Legal, x is an object
(@Shared @??? Object[]) y; // How recover element Thread-
                                                    Locality?
```

- Three solutions (at least)

  Use special annotations for arrays **COMPLEX AND BREAKS SUBSUMPTION**

  Store objects' thread-locality at run-time **EXPENSIVE W/O VM MODIFICATIONS**

  Require elements to have same thread-locality as the array **WEAKENS EXPRESSIVENESS**

- Chose the 3rd option for Loci 1.0, no reason to reconsider so far

# Example: Interfaces (& dealing with multiple inheritance)

- Default annotation for all interfaces is `@Context`, even on class level

  If class C implements interface I, it must preserve annotations

  For `@Context` annotations on I, apply viewpoint adaptation with C's annotation on LHS to obtain the correct annotation

- Interface inheritance must preserve annotations, just like for classes

```
interface I { @Shared Bar method(Foo x);  }

@Thread class C implements I {
  @Shared Bar method(@Thread Foo x) { … }
}
```

must be preserved

default elaborated in

# Shortcomings

- Coarse-grained, pragmatic system by necessity not very exact
- Some limitations on expressivity (*e.g.,* on arrays)
- Loci 1.0 has no support for transferring objects across threads

  Current solution: copy *to* shared, then *from* shared (2X overhead)

  **Good news:** code for deep-copying a class type checks using Loci annotations (this was verified for each iterative design step)

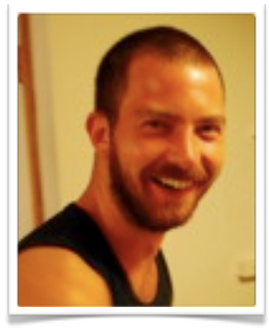
  Unclear how to solve without compromising simplicity

  Necessary for producer–consumer scenarios

- Calls to static functions lose thread-locality

  Solveable with parametric polymorphism, but solution is not perfect

# Loci 2.0



Johan Östlund
*@UU*

Francesco Zappa Nardelli
*@INRIA*

- Third thread-locality annotation

  `@ThreadSafe` (`@Thread` now **always** thread-local, better name wanted)

- Support for efficient object migration & channels

  Unique pointers to aggregate objects

  Not entirely happy with present design constraints (compromises simplicity)

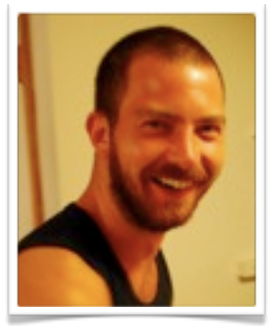
- Extend to type more properties useful for safe concurrent programming

  *e.g.*, `java.util.concurrent` libraries

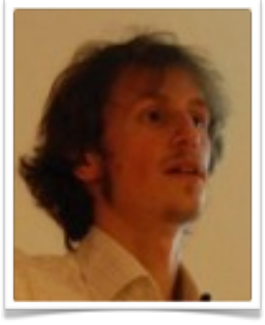
  Right now, working on ”safe fork/join parallelism”

- More evaluation—measure actual thread-locality in `@Shared` objects
- Experiment with actual thread-local heaps and parallel GC
- Improved tool support

# Loci 2.0



Johan Östlund
*@UU*



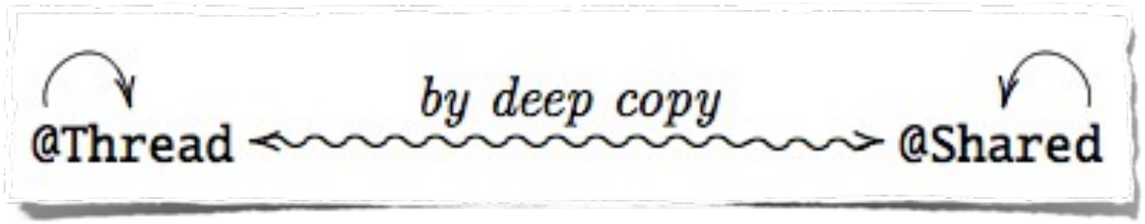
Francesco Zappa Nardelli
*@INRIA*

- Twice the annotations (–)

  More special cases, *e.g.,*

  `@ThreadSafe` not OK on fields

  But less ”magic” (+)

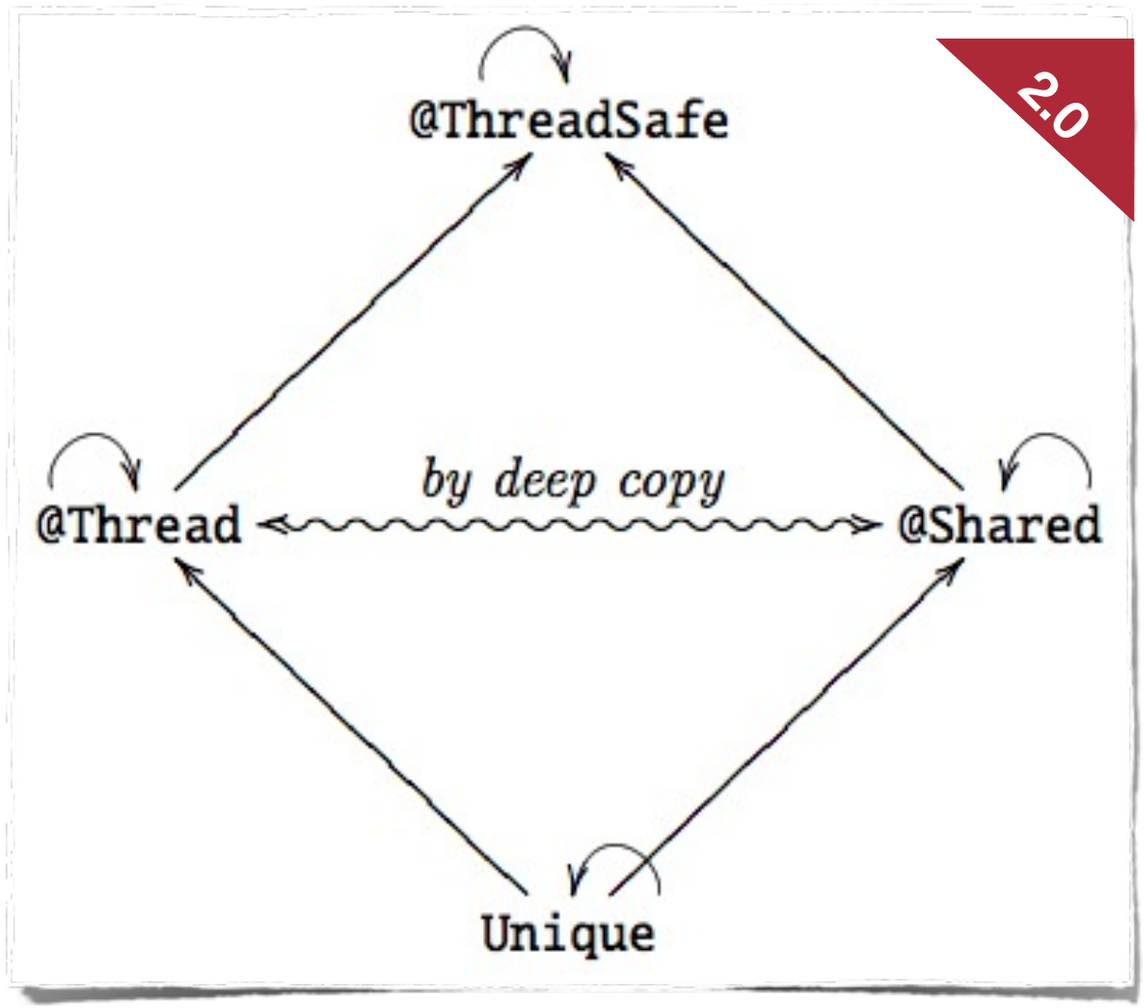



- Interesting annotation lattice

  Is it too complex to understand?

  Need more practical evaluation

  Do **YOU** have a good student?



# Conclusions

**Simple**

+ Very few rules and annotations
+ Few special cases
+ Existing Java classes work straight-off

**Compatible**

+ Most classes can be used to create thread-local objects

**Room for improvement**

– Transfer across threads, generics, etc. in Loci 2.0
– Correctness of extensions conjectured, not proven

**Experiments**

Annotated Java API is a must, more validation needed

Thank you. Questions?