Making Sequential Consistency Practical in Titanium

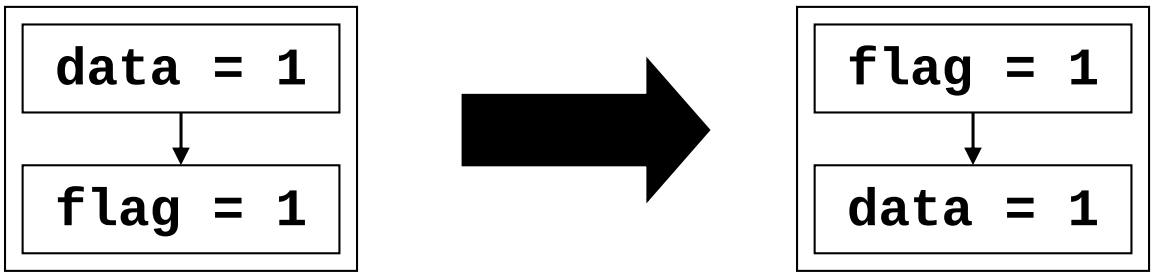
Amir Kamil, Jimmy Su, and Katherine Yelick
Titanium Group
http://titanium.cs.berkeley.edu

U.C. Berkeley
November 15, 2005

Reordering in Sequential Programs

Two accesses can be reordered as long as the reordering does not violate a local dependency.

Initially, **`flag = data = 0`**

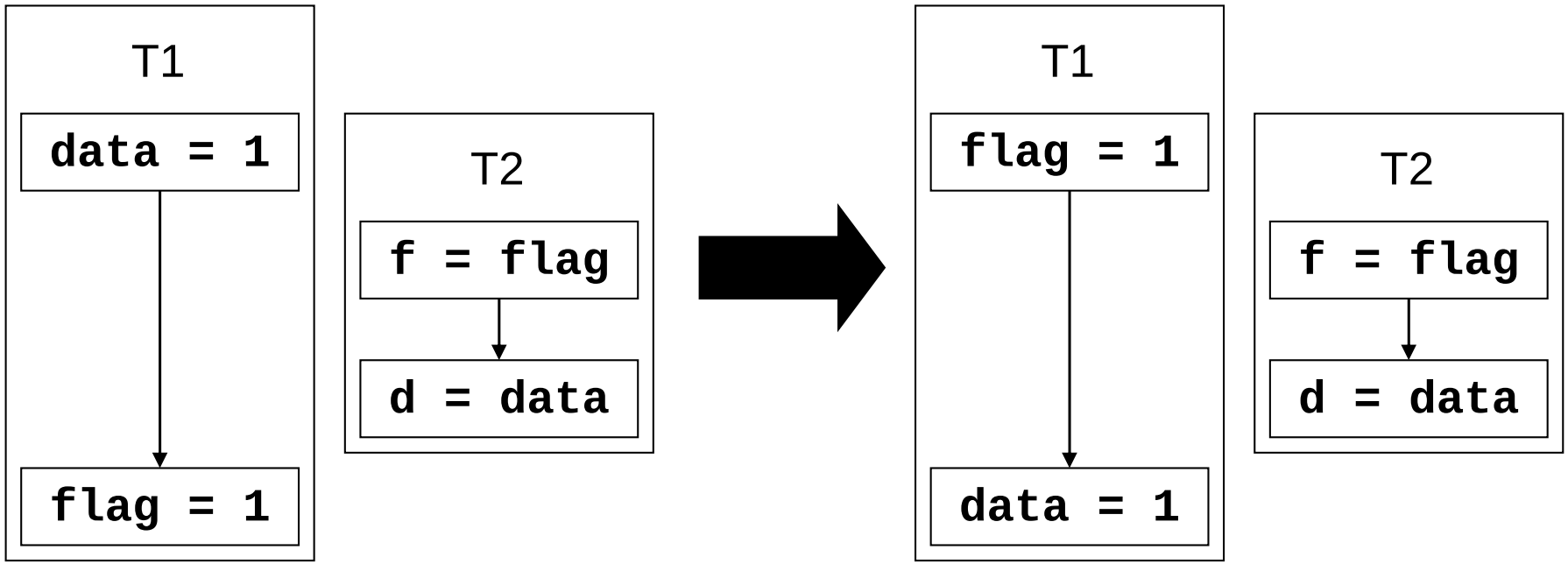


In both orderings, the end result is {`data == flag == 1`}.

Reordering in Parallel Programs

In parallel programs, a reordering can change the semantics even if no local dependencies exist.

Initially, **`flag = data = 0`**



`{f == 1, d == 0}` is a possible result in the reordered code, but not in the original code.

Memory Models

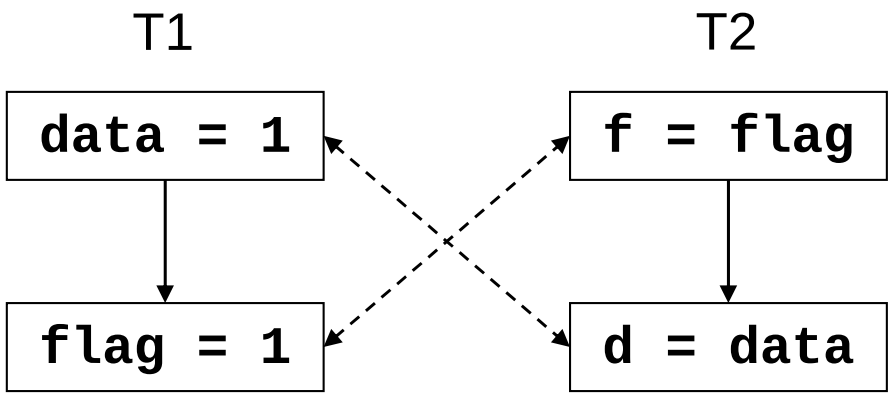
- **In *relaxed consistency*, reordering is allowed if no local dependencies or synchronization operations are violated**
- **In *sequential consistency*, a reordering is illegal if it can be observed by another thread**
- **Titanium, Java, UPC, and many other languages *do not* provide sequential consistency due to the (perceived) cost of enforcing it**

Software and Hardware Reordering

- **Compiler can reorder accesses as part of an optimization**
  - Example: copy propagation
  - *Logical fences* inserted where reordering is illegal – optimizations respect these fences
- **Hardware can reorder accesses**
  - Examples: out of order execution, remote accesses
  - *Fence instructions* inserted into generated code – waits until all prior memory operations have completed
  - Can cost a complete round trip time due to remote accesses

Conflicts

- **Reordering of an access is observable only if it *conflicts* with some other access:**
  - The accesses can be to the same memory location
  - At least one access is a write
  - The accesses can run concurrently



Conflicts

- **Fences only need to be inserted around accesses that conflict**

Sequential Consistency in Titanium

- **Minimize number of fences – allow same optimizations as relaxed model**
- **Concurrency analysis identifies concurrent accesses**
  - Relies on Titanium's *textual barriers* and *single-valued* expressions
- **Alias analysis identifies accesses to the same location**
  - Relies on SPMD nature of Titanium

Barrier Alignment

- **Many parallel languages make no attempt to ensure that barriers line up**
  - Example code that is legal but will deadlock:

```
if (Ti.thisProc() % 2 == 0)
   Ti.barrier(); // even ID threads
else
   ; // odd ID threads
```


Structural Correctness

- **Aiken and Gay introduced *structural correctness (POPL'98)***
  - Ensures that every thread executes the same number of barriers
  - Example of structurally correct code:

```
if (Ti.thisProc() % 2 == 0)
   Ti.barrier(); // even ID threads
else
   Ti.barrier(); // odd ID threads
```


Textual Barrier Alignment

- **Titanium has *textual barriers*: all threads must execute the same *textual* sequence of barriers**
  - Stronger guarantee than structural correctness – this example is illegal:

```
if (Ti.thisProc() % 2 == 0)
   Ti.barrier(); // even ID threads
else
   Ti.barrier(); // odd ID threads
```

- ***Single-valued* expressions used to enforce textual barriers**

Single-Valued Expressions

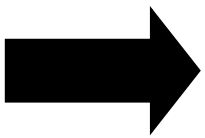
- **A *single-valued* expression has the same value on all threads when evaluated**
  - Example: `Ti.numProcs() > 1`
- **All threads guaranteed to take the same branch of a conditional guarded by a single-valued expression**
  - Only single-valued conditionals may have barriers
  - Example of legal barrier use:

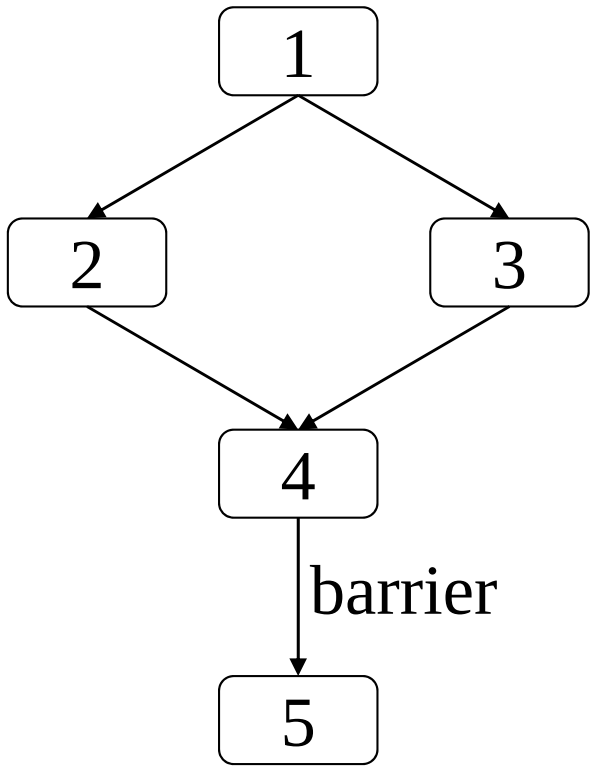
```
if (Ti.numProcs() > 1)
   Ti.barrier(); // multiple threads
else
   ; // only one thread total
```


Concurrency Analysis (I)

- **Graph generated from program as follows:**
  - Node added for each code segment between barriers and single-valued conditionals
  - Edges added to represent control flow between segments

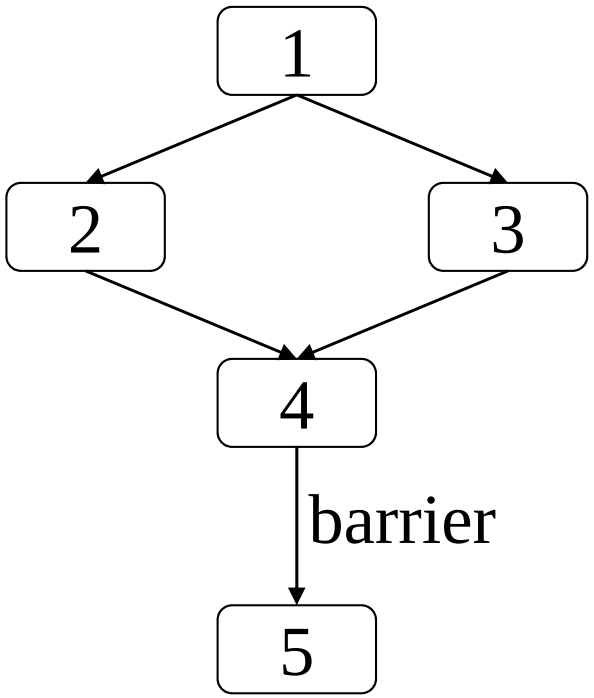
```
// code segment 1
if ([single])
  // code segment 2
else
  // code segment 3
// code segment 4
Ti.barrier()
// code segment 5
```





Concurrency Analysis (II)

- **Two accesses can run concurrently if:**
  - They are in the same node, or
  - One access's node is reachable from the other access's node without hitting a barrier
- **Algorithm: remove barrier edges, do DFS**



Concurrent Segments					
	1	**2**	**3**	**4**	**5**
1	X	X	X	X	
2	X	X		X	
3	X		X	X	
4	X	X	X	X	
5					X

Alias Analysis

- **Allocation sites correspond to *abstract locations (a-locs)***
- **All explicit and implict program variables have points-to sets**
- **A-locs are typed and have points-to sets for each field of the corresponding type**
  - Arrays have a single points-to set for all indices
- **Analysis is flow,context-insensitive**
  - Experimental call-site sensitive version – doesn't seem to help much

Thread-Aware Alias Analysis

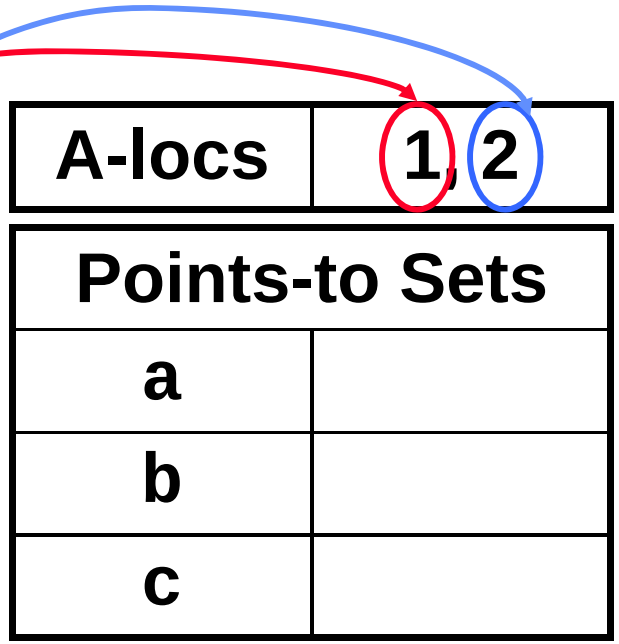
- **Two types of abstract locations: local and remote**
  - Local locations reside in local thread's memory
  - Remote locations reside on another thread
- **Exploits SPMD property**
  - Results are a summary over all threads
  - Independent of the number of threads at runtime

Alias Analysis: Allocation

- **Creates new local abstract location**
  - Result of allocation must reside in local memory

```
  class Foo {
    Object z;
  }

  static void bar() {
L1: Foo a = new Foo();
    Foo b = broadcast a from 0;
    Foo c = a;
L2: a.z = new Object();
  }
```



Alias Analysis: Assignment

- Copies source abstract locations into points-to set of target

```
  class Foo {
    Object z;
  }

  static void bar() {
L1: Foo a = new Foo();
    Foo b = broadcast a from 0;
    Foo c = a;
L2: a.z = new Object();
  }
```

A-locs	1, 2
Points-to Sets	
a	1
b	
c	1
1.z	2

Alias Analysis: Broadcast

- **Produces both local and remote versions of source abstract location**
  - Remote a-loc points to remote analog of what local a-loc points to

```
  class Foo {
    Object z;
  }

  static void bar() {
L1: Foo a = new Foo();
    Foo b = broadcast a from 0;
    Foo c = a;
L2: a.z = new Object();
  }
```

A-locs	1, 2, 1_r
Points-to Sets	
a	1
b	1, 1_r
c	1
1.z	2
1_r.z	2_r

Aliasing Results

- **Two variables A and B may *alias* if:**

  $\exists$ **x**$\in$*pointsTo*(**A**).

  x$\in$*pointsTo*(**B**)

- **Two variables A and B may *alias across threads* if:**

  $\exists$ **x**$\in$*pointsTo*(**A**).

  R(**x**)$\in$*pointsTo*(**B**),

  (where *R*(x) is the remote counterpart of x)

Points-to Sets	
a	1
b	1, 1_r
c	1

Alias [Across Threads]:	
a	b, c [b]
b	a, c [a, c]
c	a, b [b]

Benchmarks

Benchmark	Lines[1]	Description
`pi`	56	Monte Carlo integration
`demv`	122	Dense matrix-vector multiply
`sample-sort`	321	Parallel sort
`lu-fact`	420	Dense linear algebra
`3d-fft`	614	Fourier transform
`gsrb`	1090	Computational fluid dynamics kernel
`gsrb*`	1099	Slightly modified version of `gsrb`
`spmv`	1493	Sparse matrix-vector multiply
`gas`	8841	Hyperbolic solver for gas dynamics

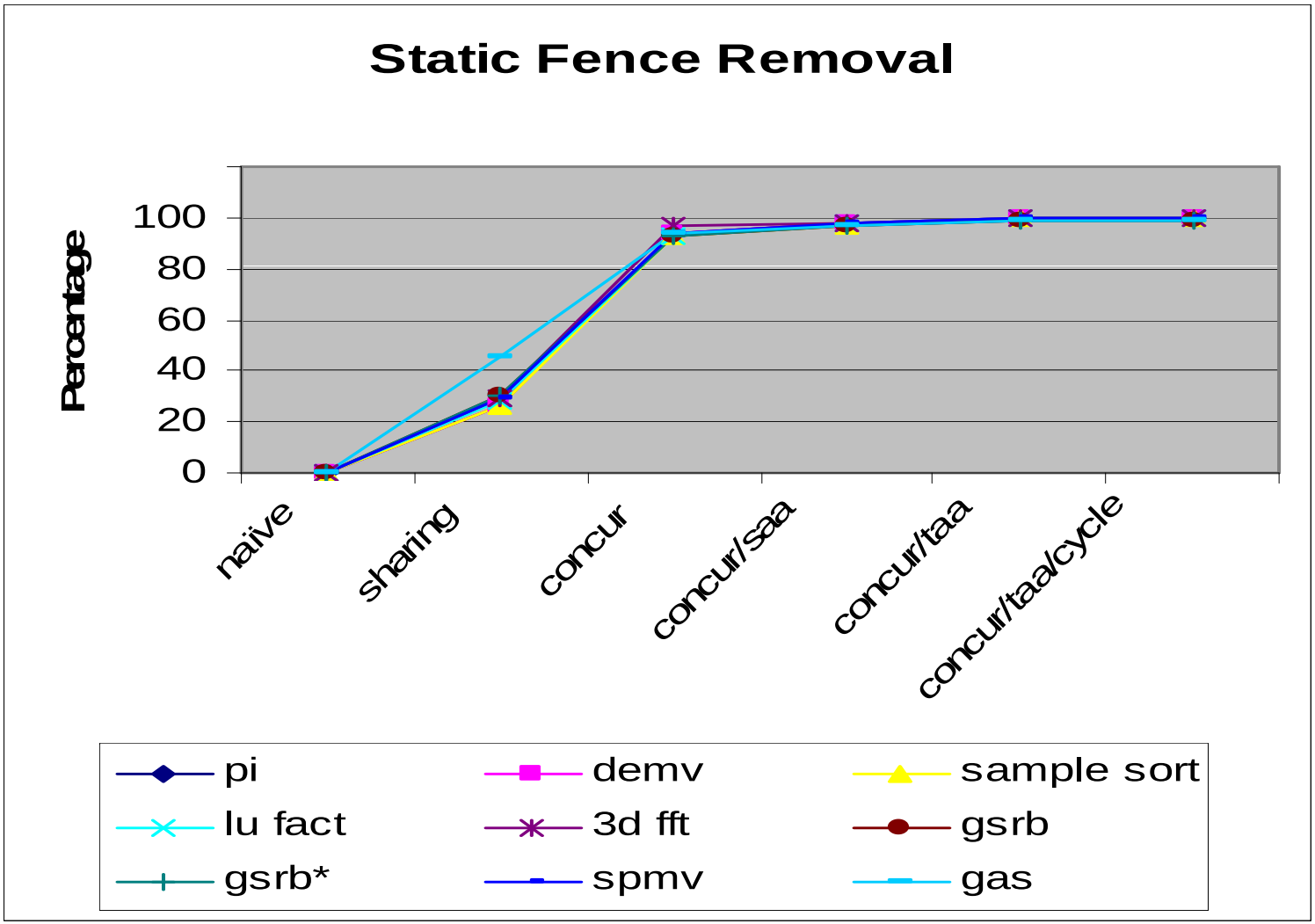
[1] Line counts do not include the reachable portion of the 37,000 line Titanium/Java 1.0 libraries

Analysis Levels

- **We tested analyses of varying levels of precision**

Analysis	Description
`naïve`	All heap accesses
`sharing`	All shared accesses
`concur`	Concurrency analysis + type-based AA
`concur/saa`	Concurrency analysis + sequential AA
`concur/taa`	Concurrency analysis + thread-aware AA
`concur/taa/cycle`	Concurrency analysis + thread-aware AA + cycle detection

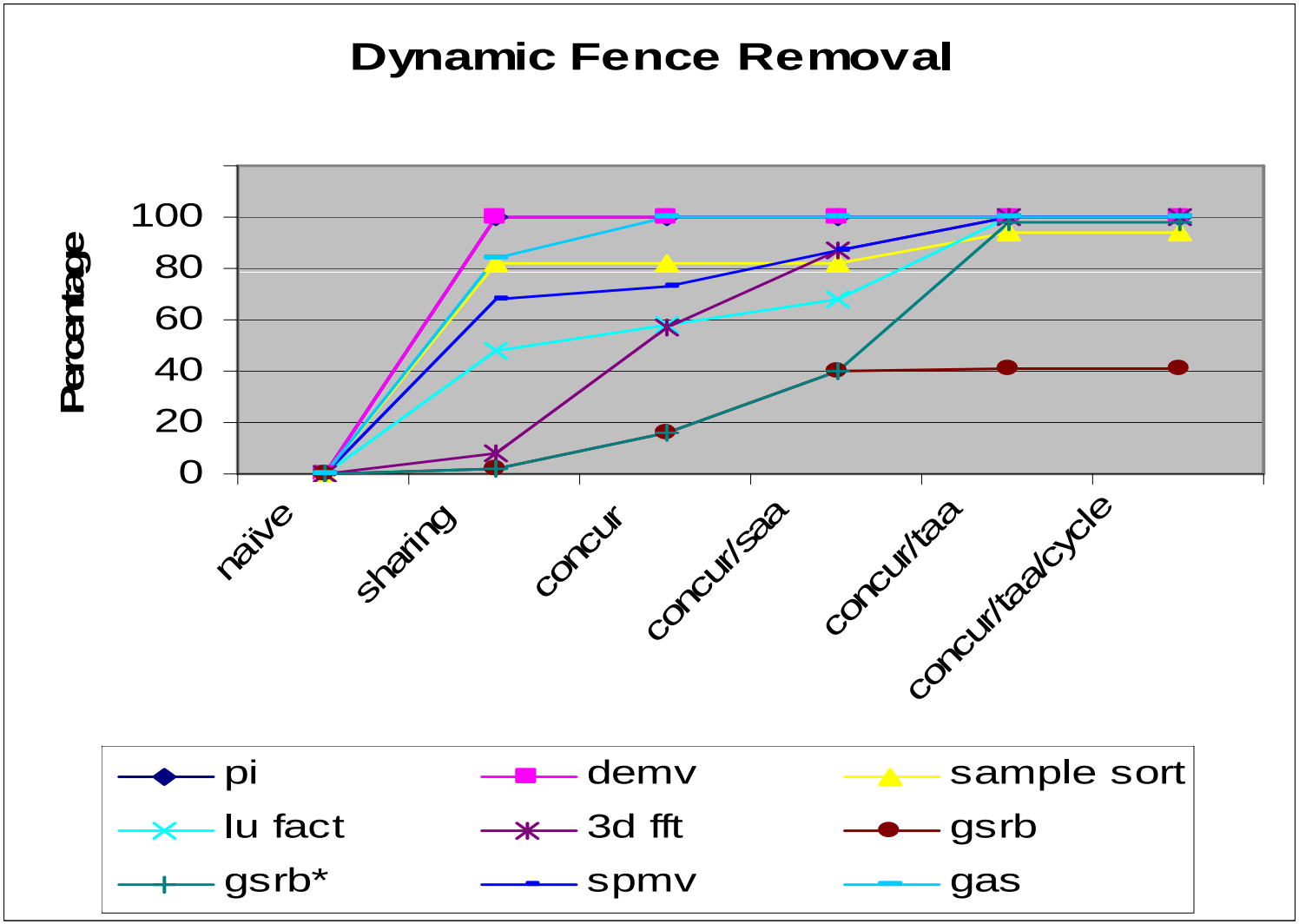
Static (Logical) Fences



GOOD

Percentages are for number of static fences reduced over `naive`

Dynamic (Executed) Fences

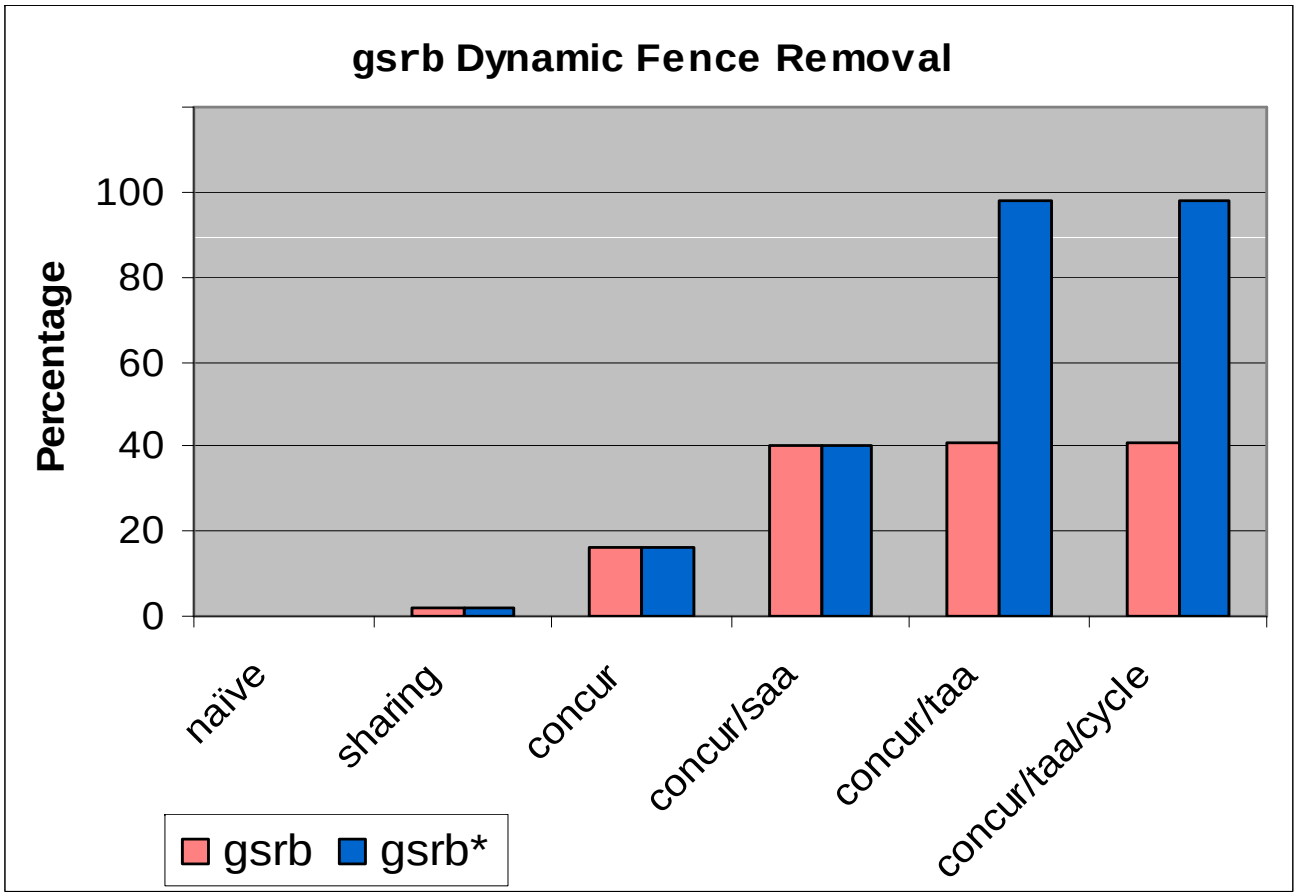


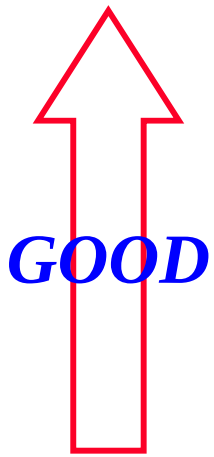
GOOD

Percentages are for number of dynamic fences reduced over `naive`

Dynamic Fences: `gsrb`

- **`gsrb` relies on dynamic locality checks**
  - slight modification to remove checks (**`gsrb*`**) greatly increases precision of analysis





Two Example Optimizations

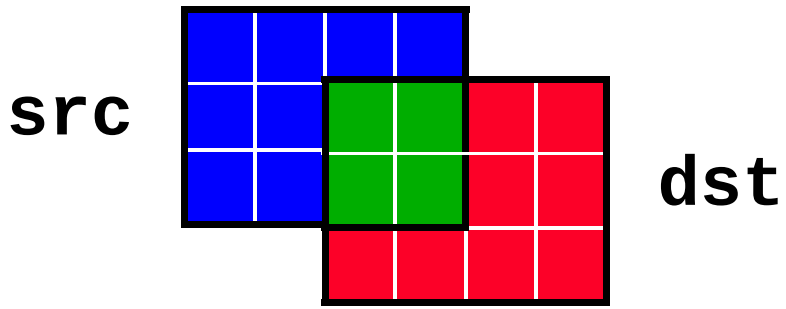
- **Consider two optimizations for GAS languages**
  1. Overlap bulk memory copies
  2. Communication aggregation for irregular array accesses (i.e. `a[b[i]]`)
- **Both optimizations reorder accesses, so sequential consistency can inhibit them**
- **Both are addressing network performance, so potential payoff is high**

Array Copies in Titanium

- **Array copy operations are commonly used**

```
dst.copy(src);
```

- **Content in the domain intersection of the two arrays is copied from `dst` to `src`**



- **Communication (possibly with packing) required if arrays reside on different threads**
- **Processor blocks until the operation is complete.**

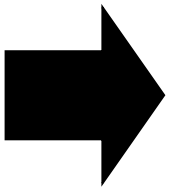
Non-Blocking Array Copy Optimization

- **Automatically convert blocking array copies into non-blocking array copies**
  - Push sync as far down the instruction stream as possible to allow overlap with computation
- ***Interprocedural:* syncs can be moved across method boundaries**
- **Optimization reorders memory accesses – may be illegal under sequential consistency**

Communication Aggregation on Irregular Array Accesses (Inspector/Executor)

- **A loop containing indirect array accesses is split into phases**
  - *Inspector* examines loop and computes reference targets
  - Required remote data gathered in a bulk operation
  - *Executor* uses data to perform actual computation

```
for (...) {
  a[i] = remote[b[i]];
  // other accesses
}
```



```
schd = inspect(remote, b);
tmp = get(remote, schd);
for (...) {
  a[i] = tmp[i];
  // other accesses
}
```

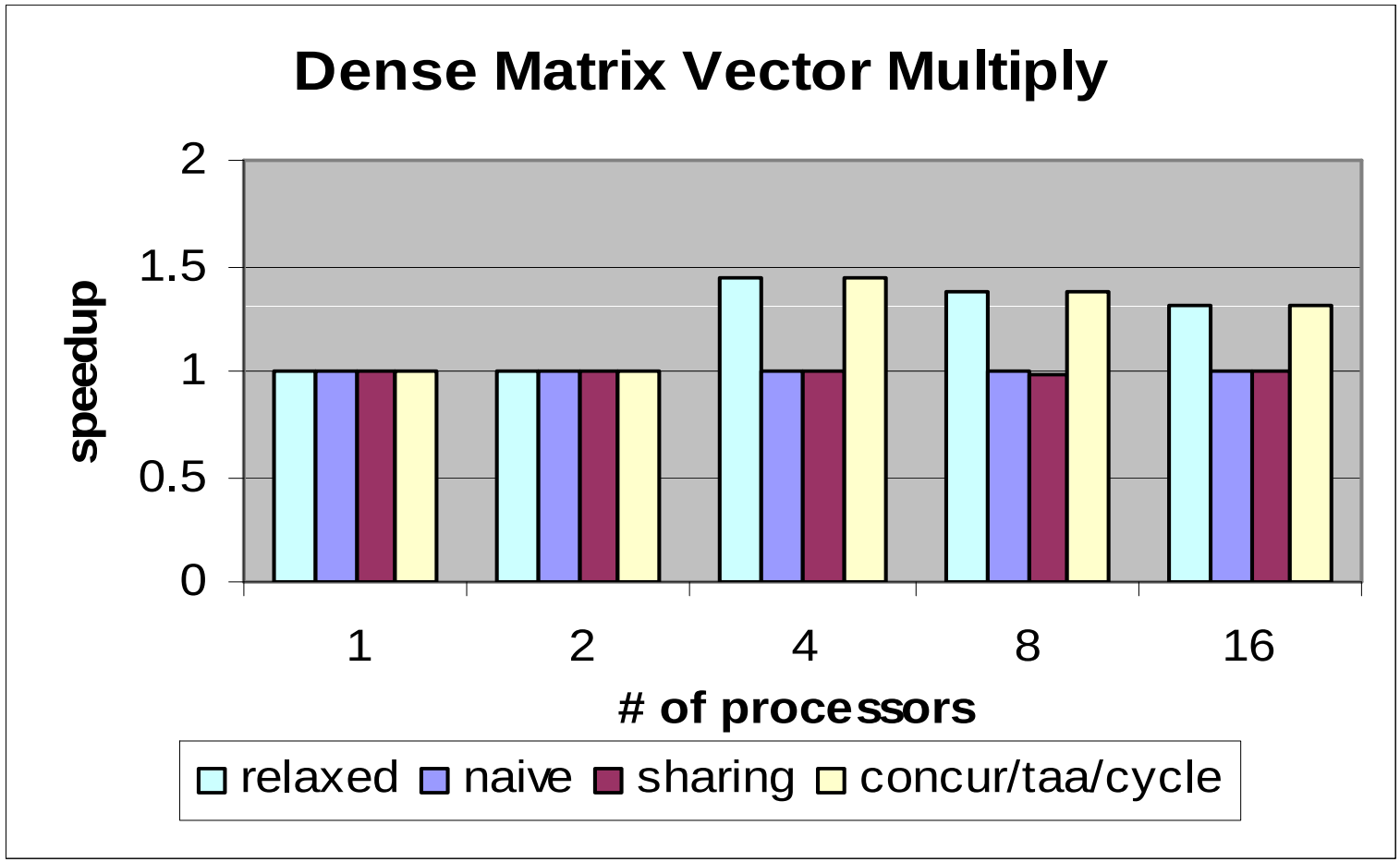
- **Can be illegal under sequential consistency**

Relaxed + SC with 3 Analyses

- **We tested performance using analyses of varying levels of precision**

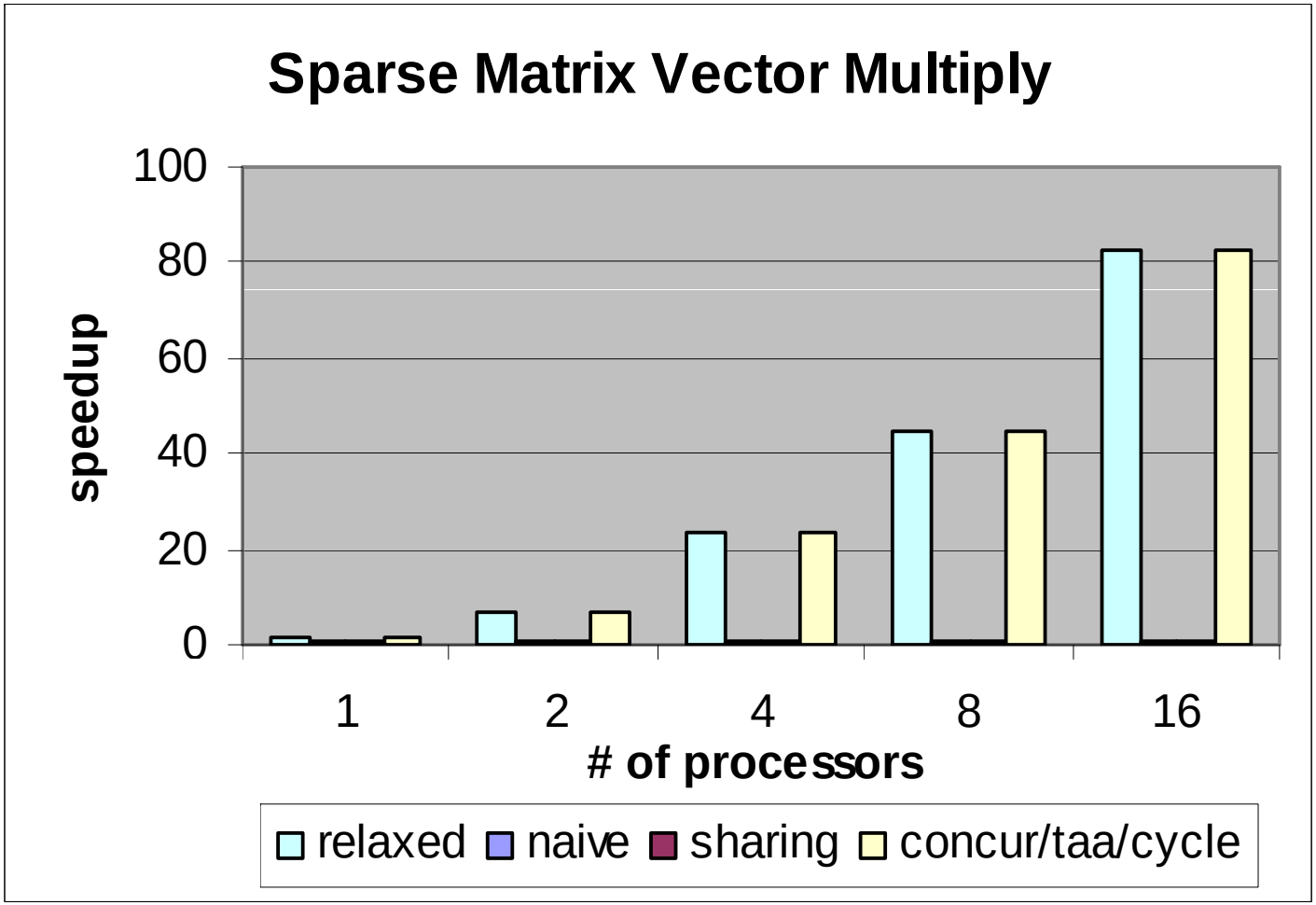
Name	Description
`relaxed`	Uses Titanium's relaxed memory model
`naïve`	Uses sequential consistency, puts fences around every heap access
`sharing`	Uses sequential consistency, puts fences around every shared heap access
`concur/taa/cycle`	Uses sequential consistency, uses our most aggressive analysis

Dense Matrix Vector Multiply



- Non-blocking array copy optimization applied
- Strongest analysis is necessary: other SC implementations suffer relative to relaxed

Sparse Matrix Vector Multiply



- Inspector/executor optimization applied
- Strongest analysis is again necessary and sufficient

Conclusion

- **Titanium's textual barriers and single-valued expressions allow for simple but precise concurrency analysis**
- **Sequential consistency can eliminate nearly all fences for the benchmarks tested**
- **On two linear algebra kernels, sequential consistency can be provided with little or no performance cost with our analysis**
  - Analysis allows the same optimizations to be performed as in the relaxed memory model