CACHING IN ON AGGREGATION

image credit: UK National Archives

Michael Ferguson
mferguson@ltsnet.net

MEMORY MODEL BACKGROUND

Memory model for

`C11`, **`C++11`**, **`Chapel`**:

data race free programs are sequentially consistent

- See Adve, S.V., Boehm, H.-J. 2010. Memory models: a case for rethinking parallel languages and hardware. Communications of the ACM 53(8): 90–101. http://cacm.acm.org/magazines/2010/8/96610-memory-models-a-case-for-rethinking-parallel-languages-and-hardware/fulltext

A RACY PROGRAM

Thread 1

```
x = f();
done = true;
```

Thread 2

```
while(!done) { }
print(x);
```

A RACY PROGRAM

Thread 1

```
x = f();
done = true;
```

Thread 2

```
while(!done) { }
print(x);
```

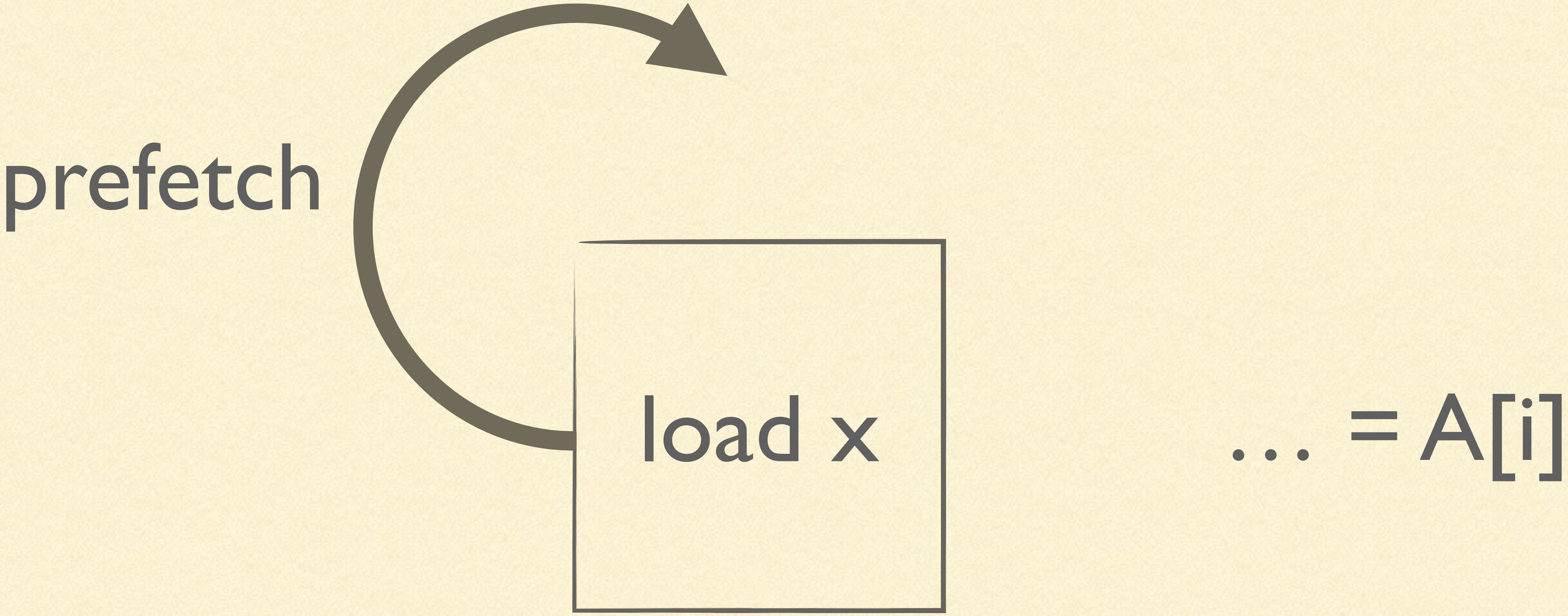
compiler *or* processor

Thread 1

```
r1 = f();
done = true; x = r1;
```

Thread 2

```
r1 = done; while(!r1) { }
print(x);
```



prefetch

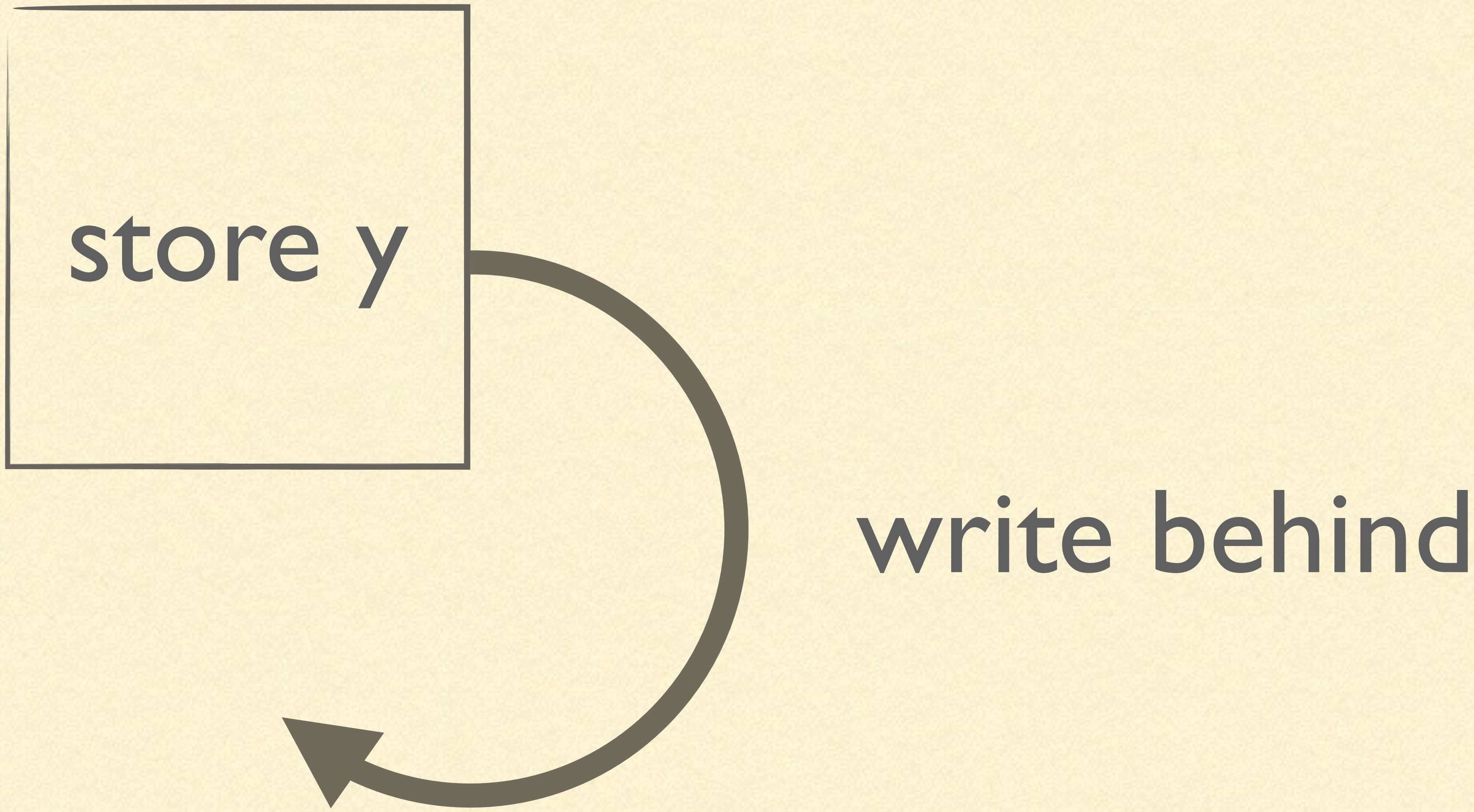
load x

… = A[i]

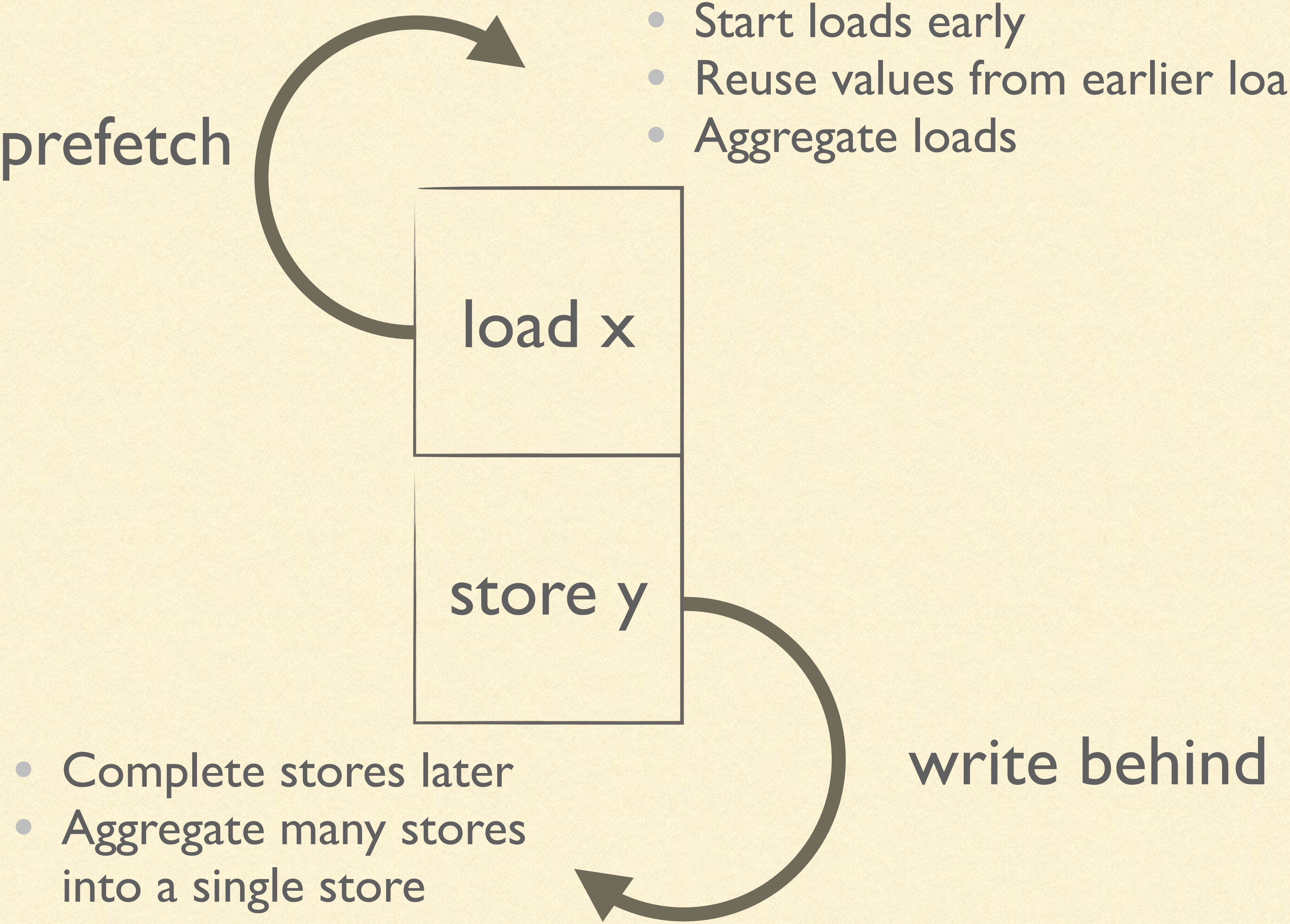
Compiler *and* processor would like to start loads earlier in order to hide memory latency. We’ll call that *prefetch*.

Compiler *and* processor would like to complete stores later in order to hide memory latency. We’ll call that *write behind*.

B[i] = …



- Start loads early
- Reuse values from earlier load
- Aggregate loads



- Complete stores later
- Aggregate many stores into a single store

REMEMBER THE RACY PROGRAM?

Thread 1

```
x = f();
done = true;
```

Thread 2

```
while(!done) { }
print(x);
```

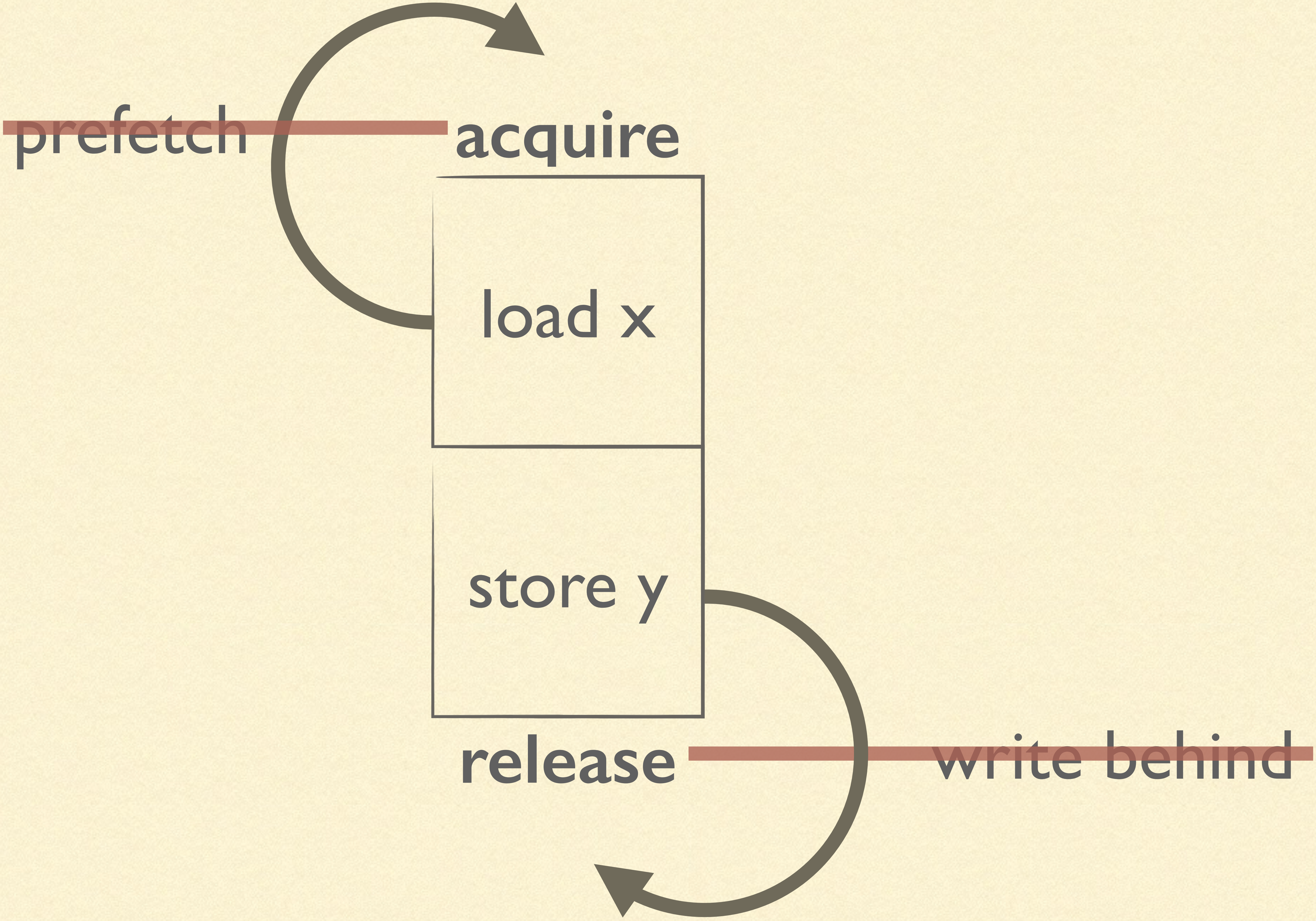
compiler *or* processor

Thread 1

```
r1 = f();
done = true; x = r1;
```

Thread 2

```
r1 = done; while(!r1) { }
print(x);
```



~~prefetch~~

acquire

load x

store y

release

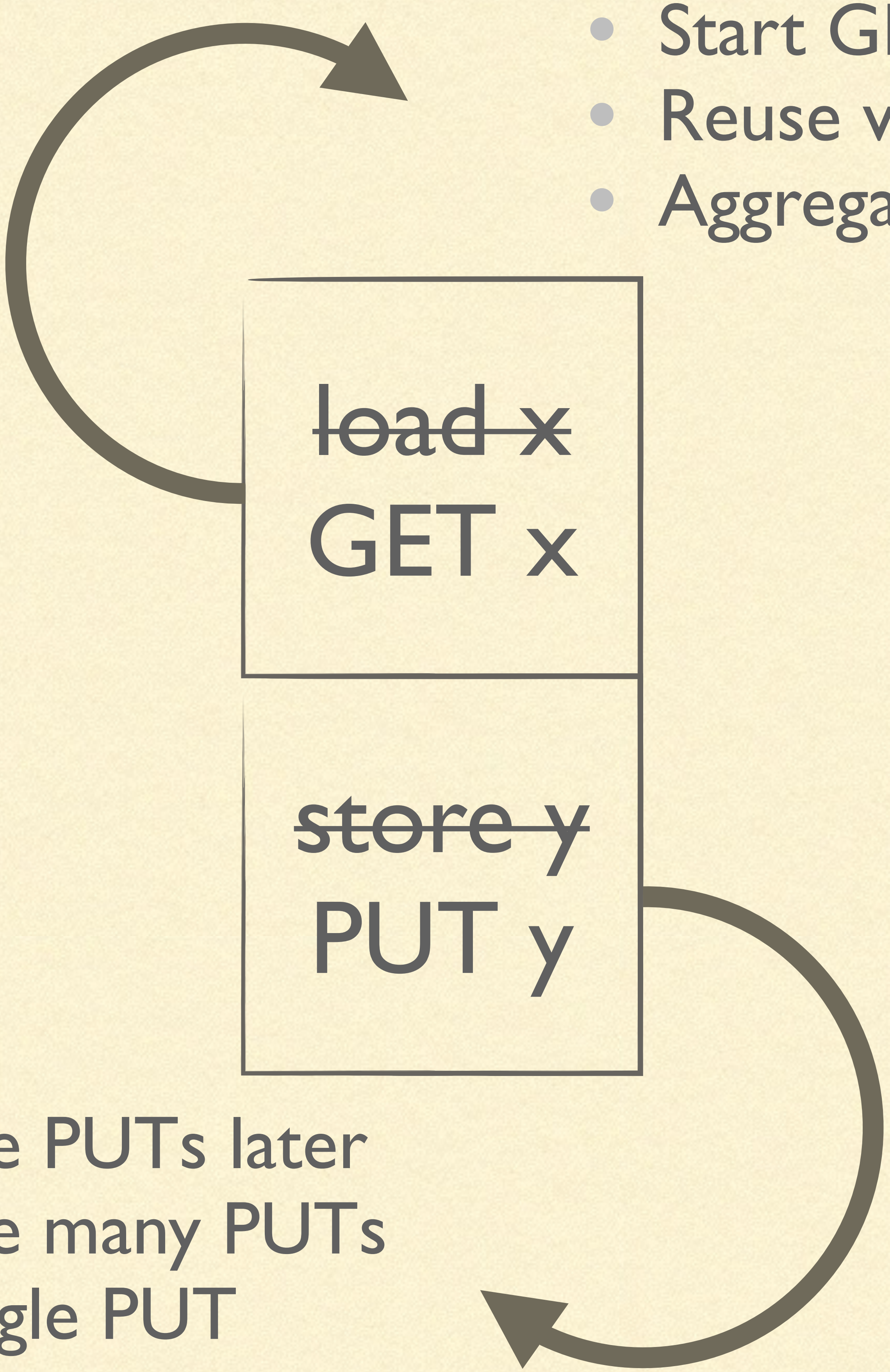
~~write behind~~

COMMUNICATION OPTIMIZATION

Library of Congress

- Start GETs early
- Reuse values from earlier GET
- Aggregate GETs

prefetch



~~load x~~
GET x

~~store y~~
PUT y

write behind

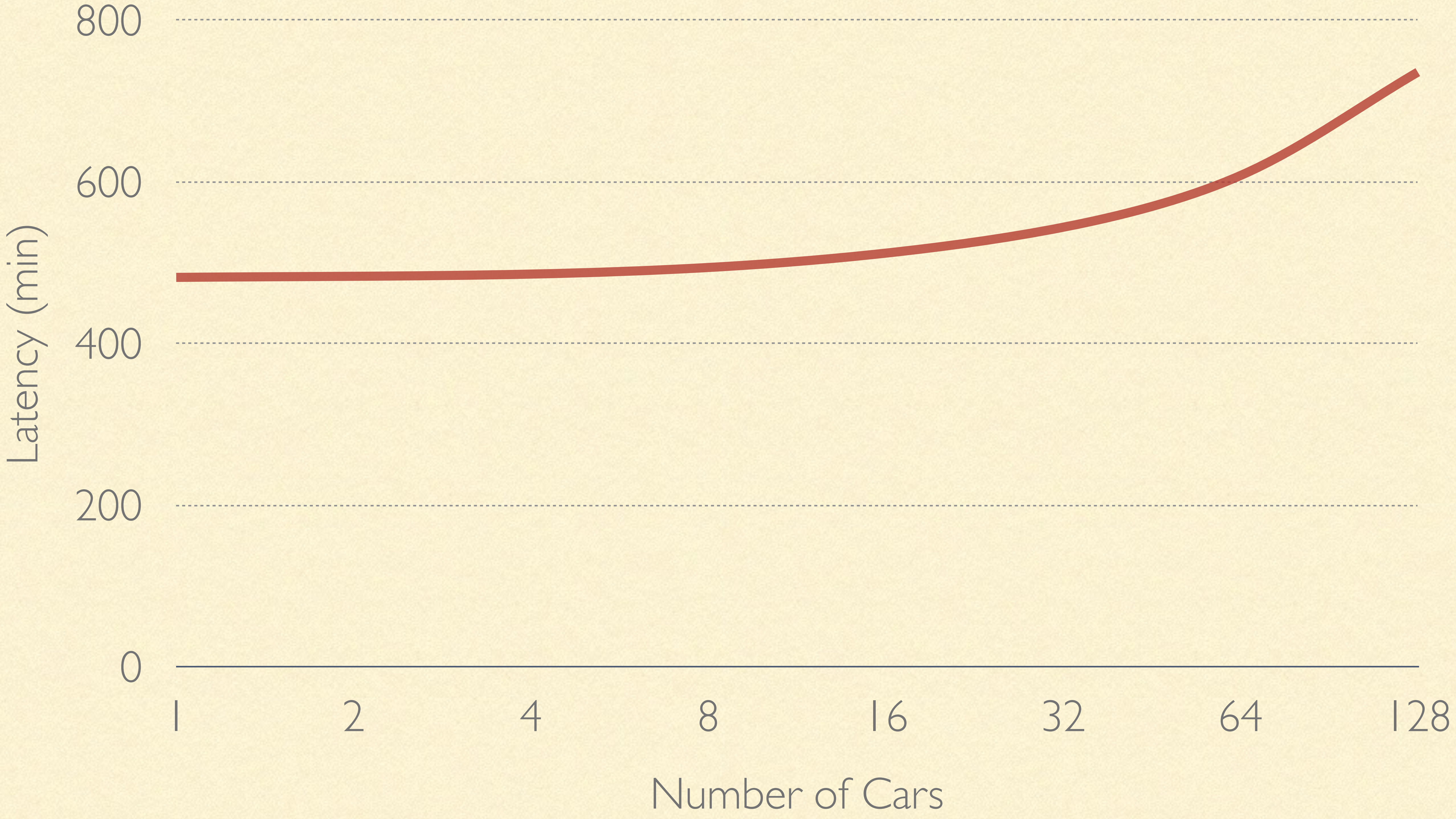
- Complete PUTs later
- Aggregate many PUTs into a single PUT

CC Flickr/Daniel Jolivet

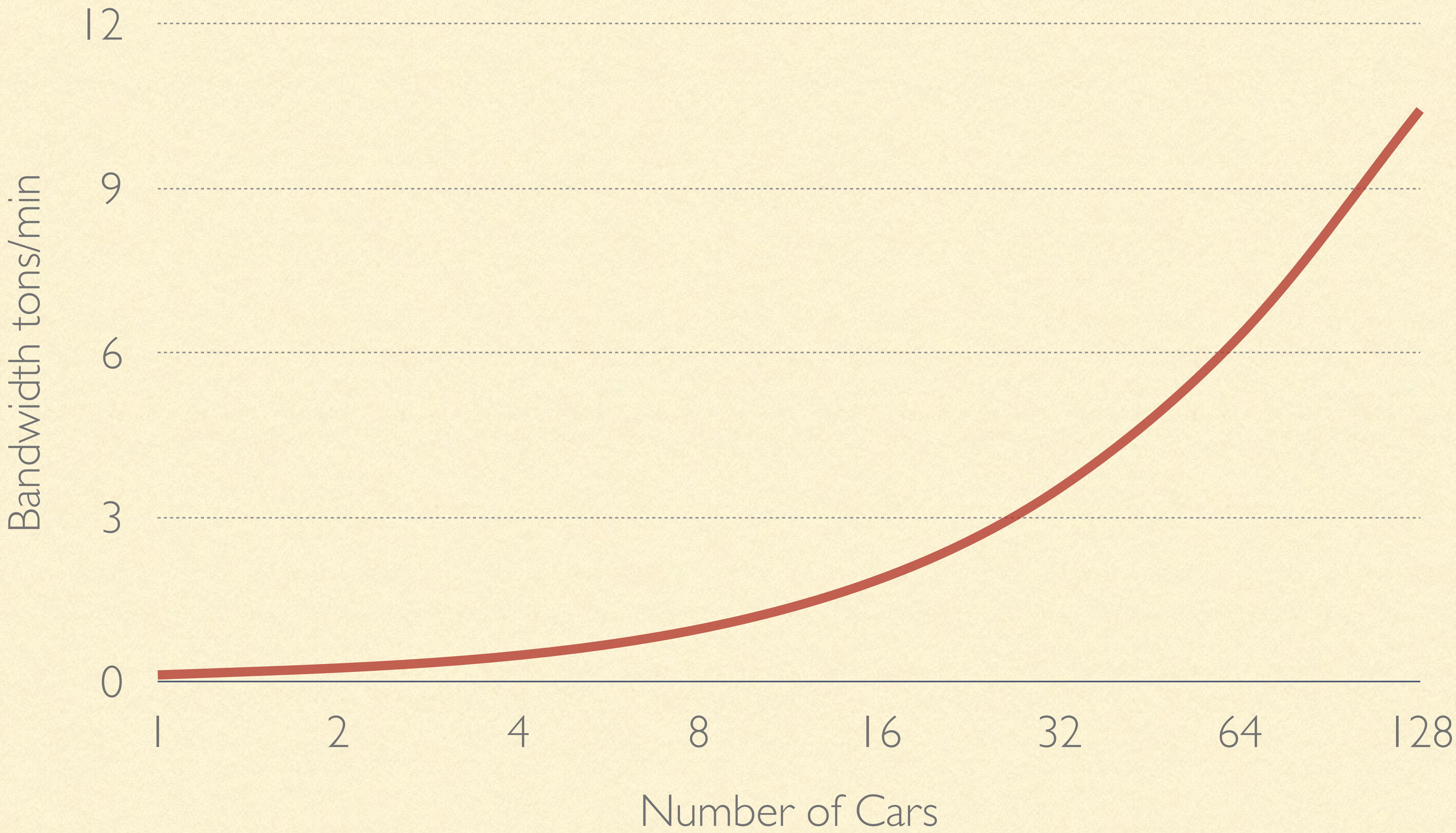
CC Flickr/Ben Salter

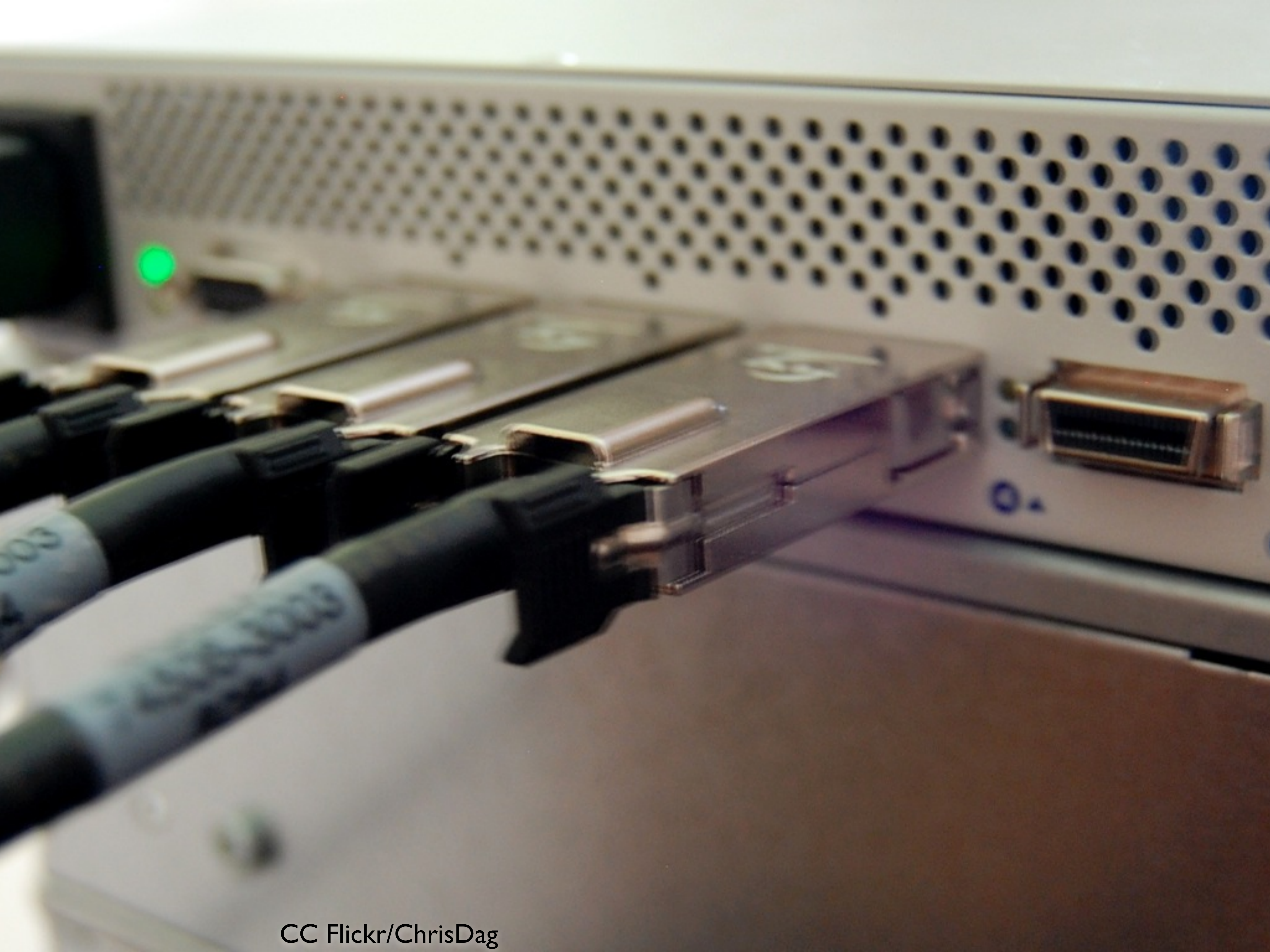
CC Flickr/ajmexico

TRAIN LATENCY (8 HOUR TRIP, 60 TON CARS, 60 SEC/CAR)



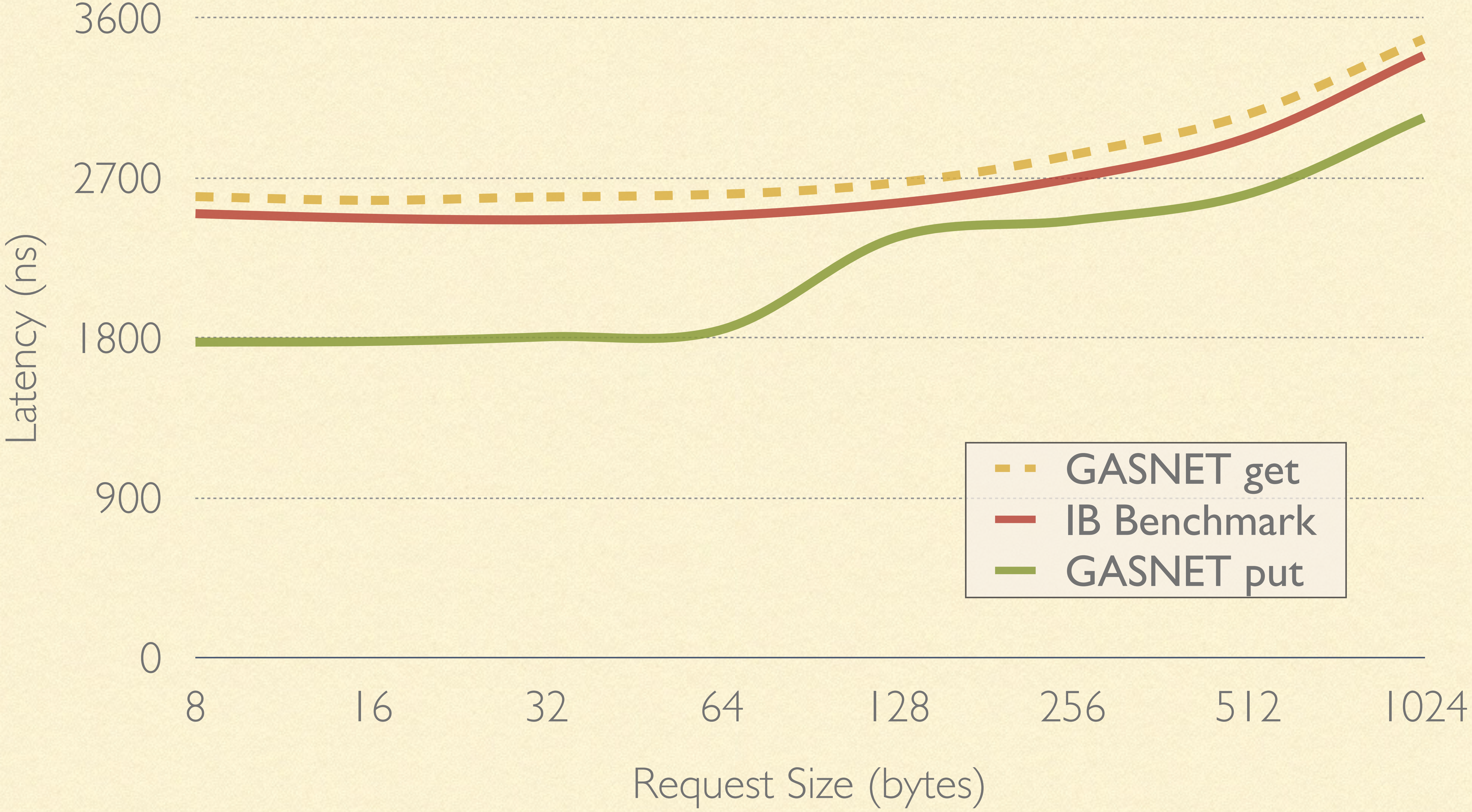
TRAIN BANDWIDTH



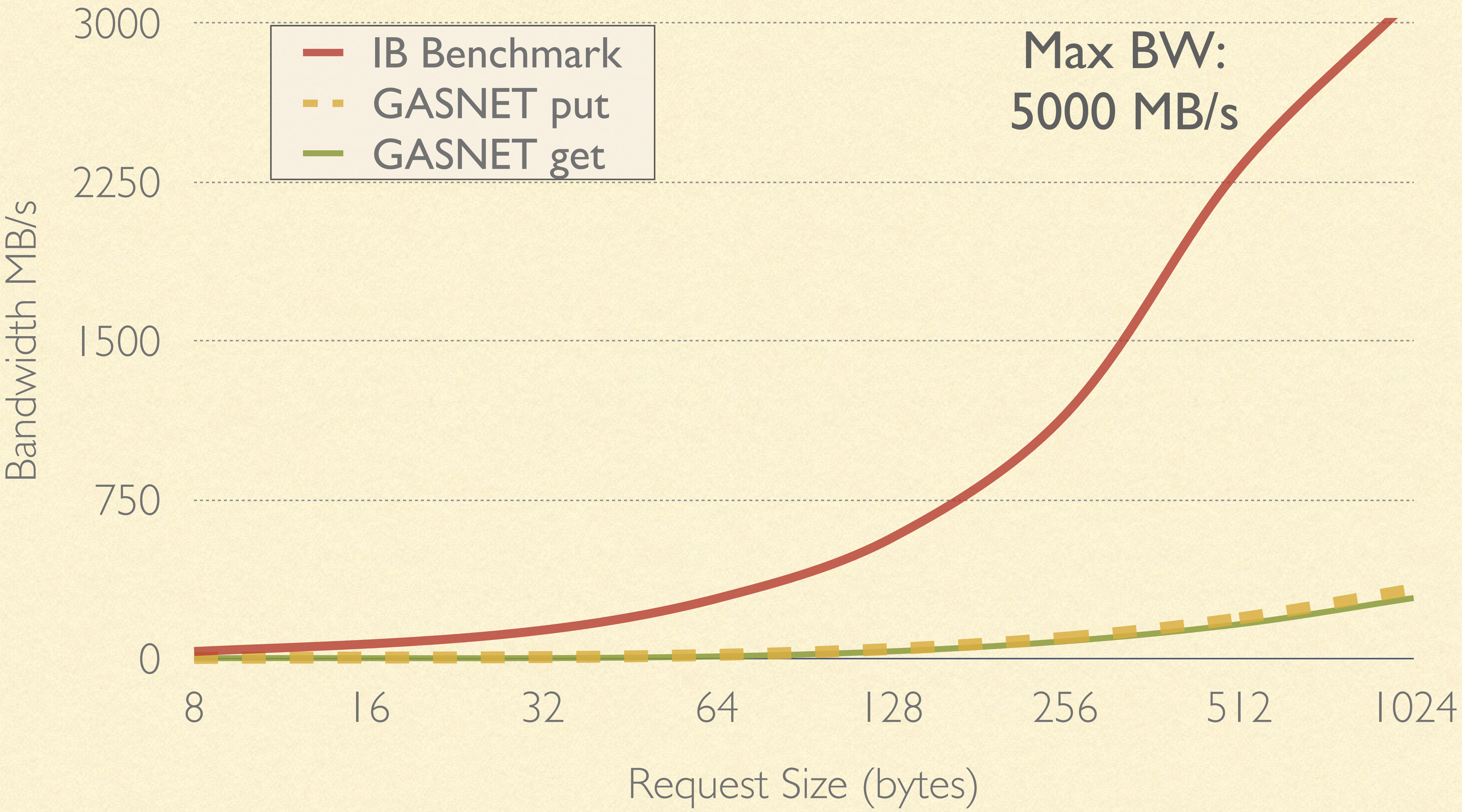


CC Flickr/ChrisDag

INFINIBAND (IB) LATENCY

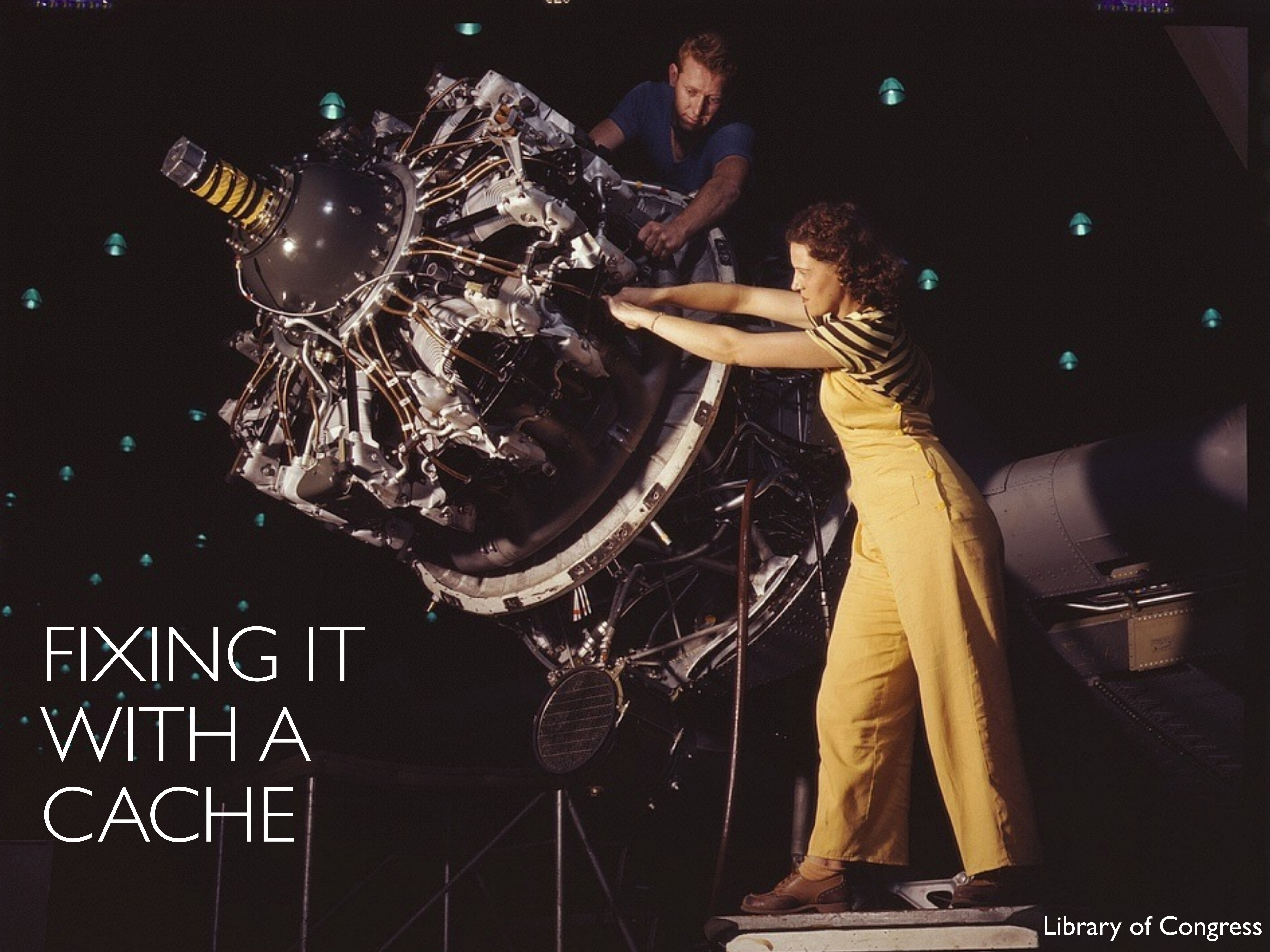


INFINIBAND (IB) BANDWIDTH



Max BW:
5000 MB/s

FIXING IT WITH A CACHE



Library of Congress

NO COHERENCY TRAFFIC

- Avoid a noisy coherency protocol
- Aggregation, prefetch, and write-behind still work
- Discard all cached data on *acquire*
- Wait for pending operations on a *release*

ADDING IMPLIED FENCES

- *acquire* and *release* triggered by task or on statement spawn, join, start, and finish

```
release
on {
  acquire
  ...
  release
}
acquire
```

```
sync {
  release
  begin {
    acquire
    ....
    release
  }
} acquire
```

CACHE PER PTHREAD

- Too hot: 1 cache per locale
  - complex implementation, slow locks, etc
- Too cold: 1 cache per task
  - cache is probably bigger than task stack
- Just right: 1 cache per pthread/core
  - easy to implement with pthread-local storage

OTHER DESIGN NOTES

- Allocates all cache memory only once
    - malloc takes ~1μs … infiniband latency is ~2μs!
- Reads entire 64-byte cache-line at a time
- Automatic write-behind and sequential read ahead
- User-operable *prefetch* hint

USABILITY

Library of Congress

COPY EXAMPLE

```
var A:[1..n] int;
var B:[1..n] int;
on Locales[1] {
  for i in 1..n {
    B[i] = A[i];
  }
}
```

… = A[i] is a GET
B[i] = … is a PUT

=> n GETs *
n PUTs

* 5n GETs currently because of array header loads

MESSY EXPLICIT AGGREGATION

```
var A:[1..n] int;
var B:[1..n] int;
on Locales[1] {
  for i in 1..n by k{
    B[i..k]=A[i..k];
  } …
}
```

- Array slices currently very heavy-weight
- k depends on hardware, not problem
- Tricky boundaries

PREFETCH EXAMPLE

```
var A:[1..n] int;
on Locales[1] {
 var sum:int;
 for i in 1..n {
  prefetch(A[f(i+k)]);
  sum += A[f(i)];
 }
}
```

prefetch(…) is a prefetch hint

- just like cache optimization
- no awkward handles

AWKWARD HANDLES?

```
var A:[1..n] int;
on Locales[1] {
 var sum:int;
 for i in 1..n {
  prefetch(A[f(i+k)]);
  sum += A[f(i)];
 }
}
```

```
var A:[1..n] int;
on Locales[1] {
 var sum:int;
 var h[1..k]:…;
 for i in 1..n {
  h[…] = get_nb(A[f(i+k)])
  sum += wait(h[…]);
 } …
}
```

BENCHMARKS

San Diego Air and Space Museum

COPY EXAMPLE

```
var A:[1..n] int;
var B:[1..n] int;
on Locales[1] {
  for i in 1..n {
    B[i] = A[i];
  }
}
```

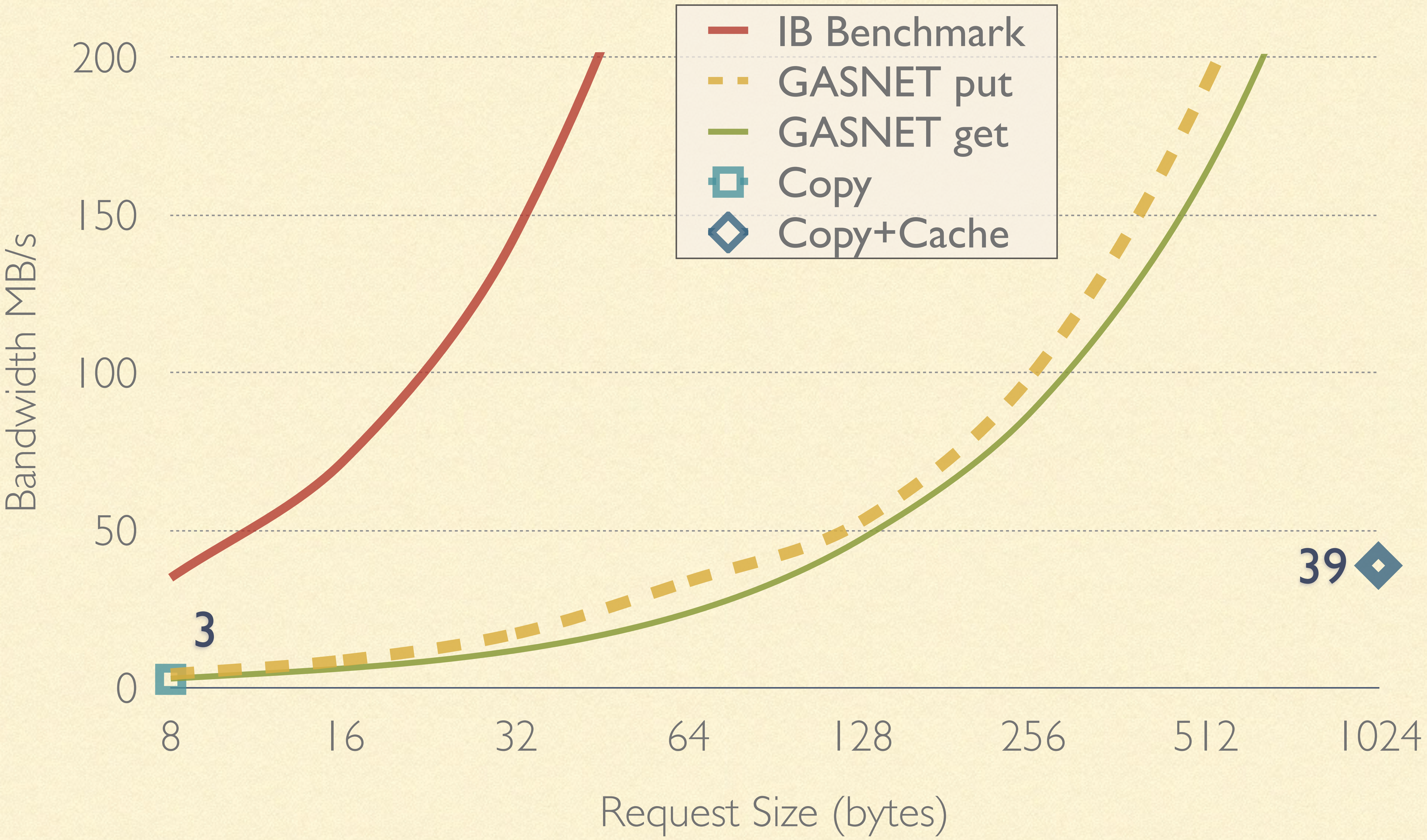
… = A[i] is a GET
and done in chunks of
1024 bytes with readahead

B[i] = … is a PUT
and done in chunks of
1024 bytes with write-behind

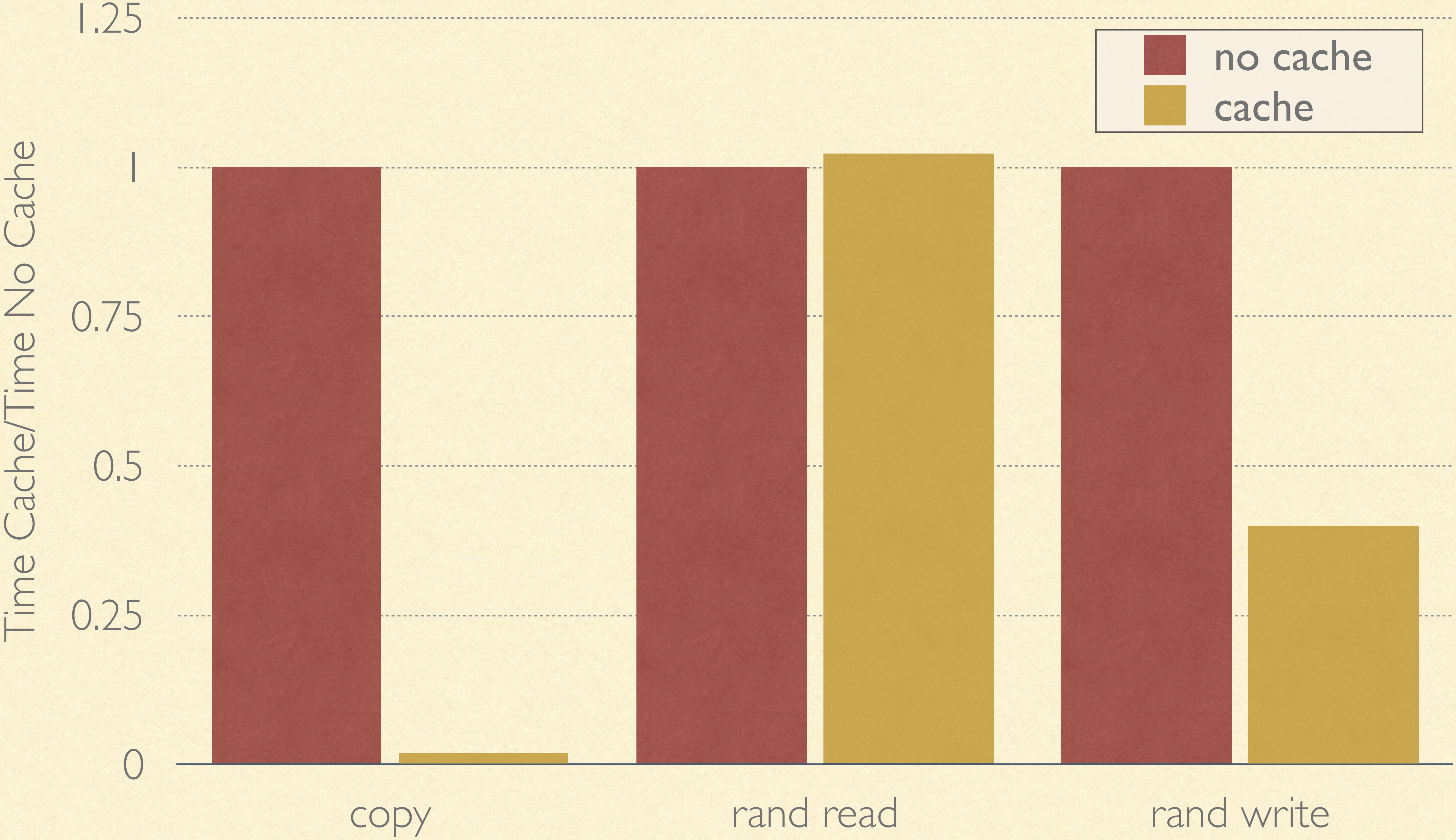
array header overhead removed

→ **56x speedup!**

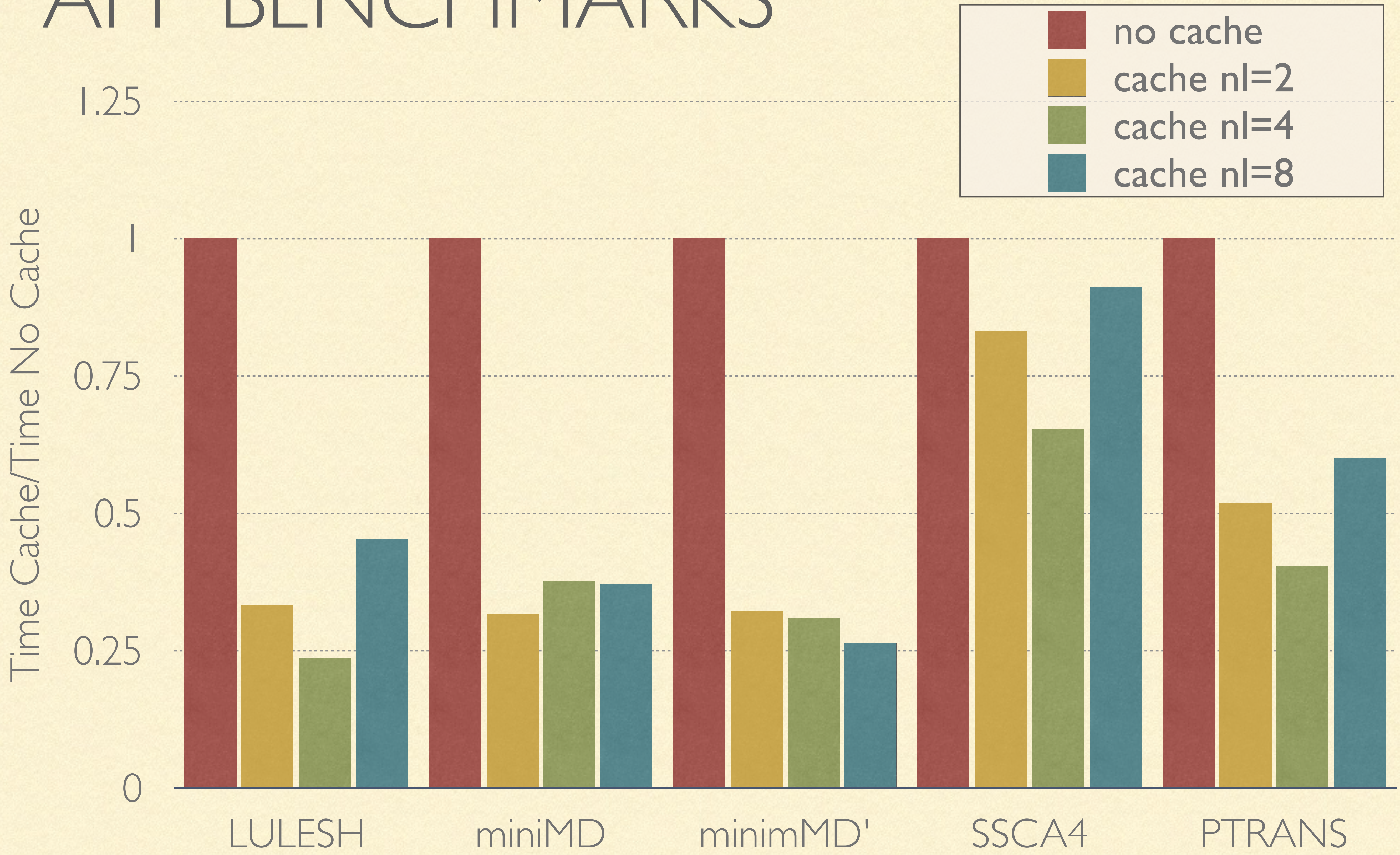
EXAMPLE PERFORMANCE



SYNTHETIC BENCHMARKS

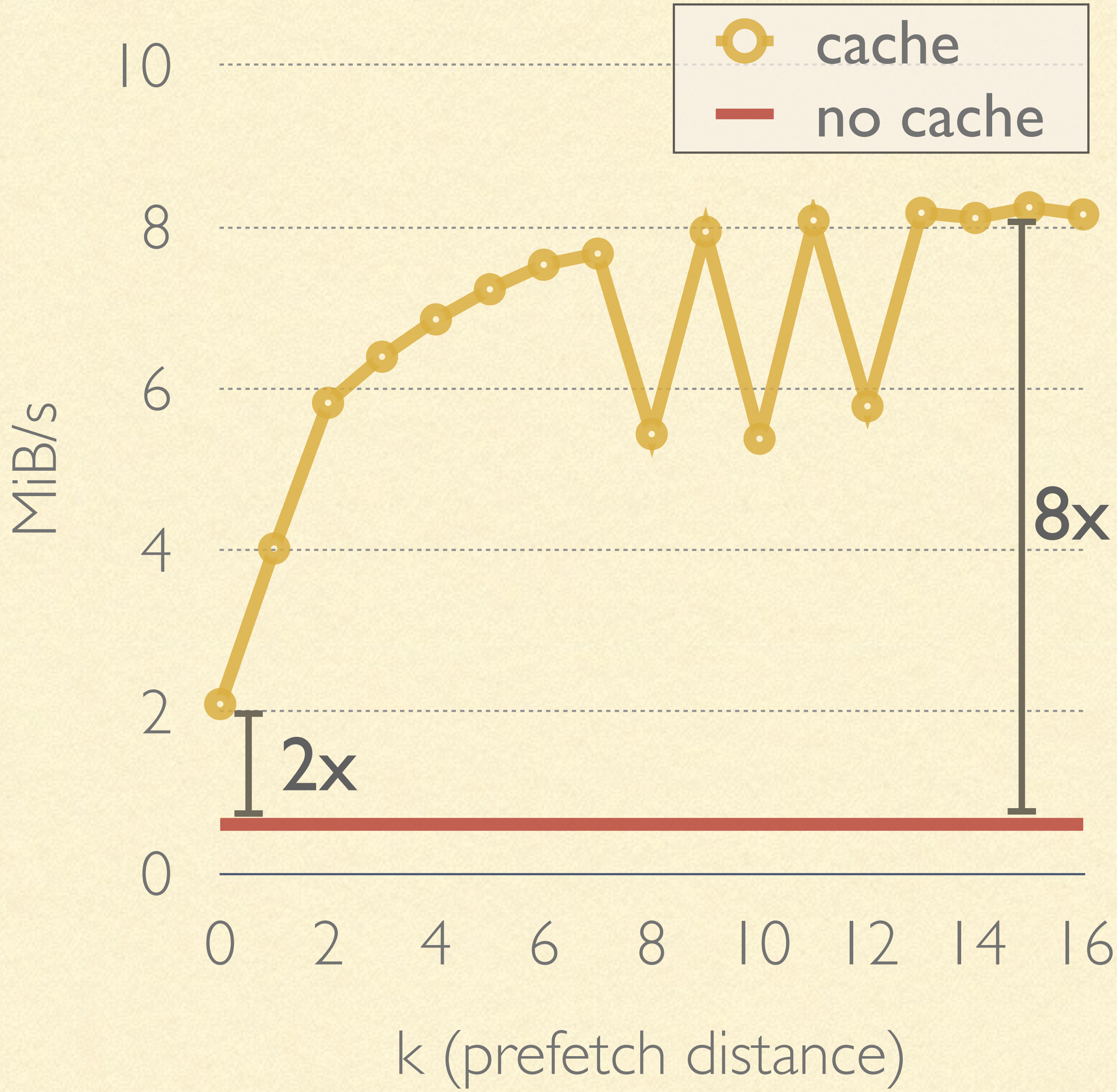


APP BENCHMARKS



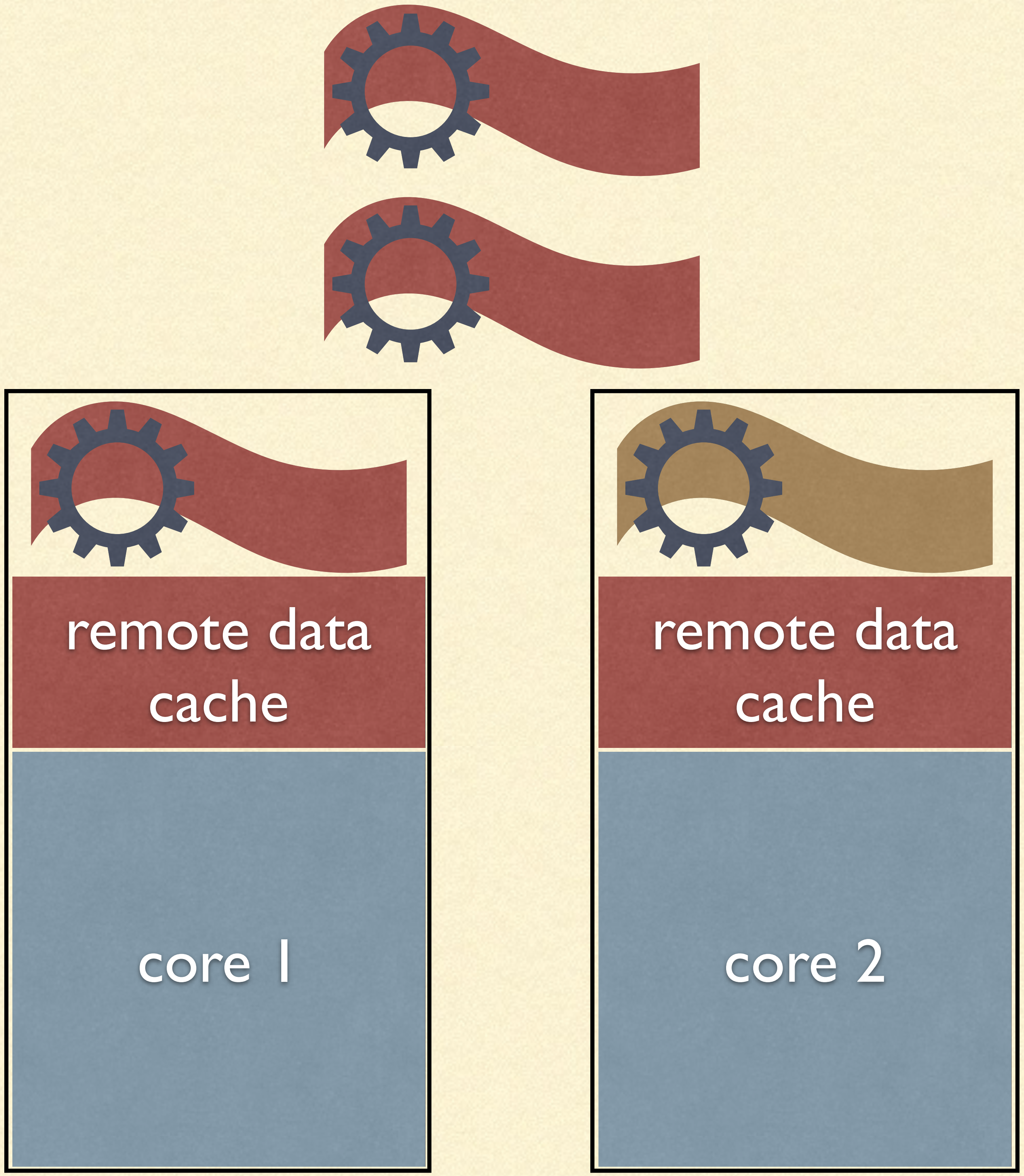
PREFETCH EXAMPLE

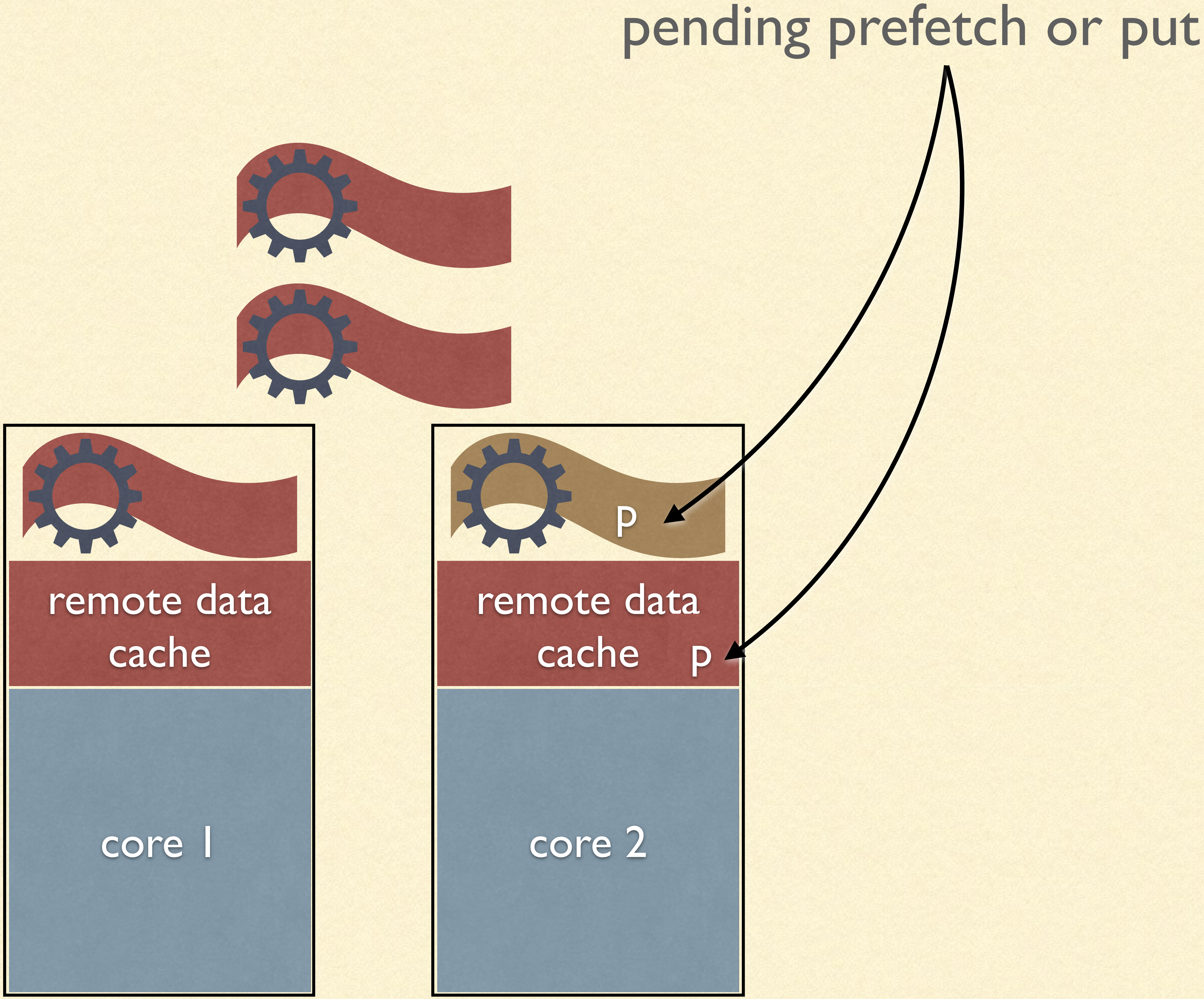
```
var A:[1..n] int;
on Locales[1] {
 var sum:int;
 for i in 1..n {
  prefetch(A[f(i+k)]);
  sum += A[f(i)];
 }
}
```



TASKING TROUBLE

George Eastman House Collection


remote data cache
core 1
remote data cache
core 2



pending prefetch or put

p

remote data cache

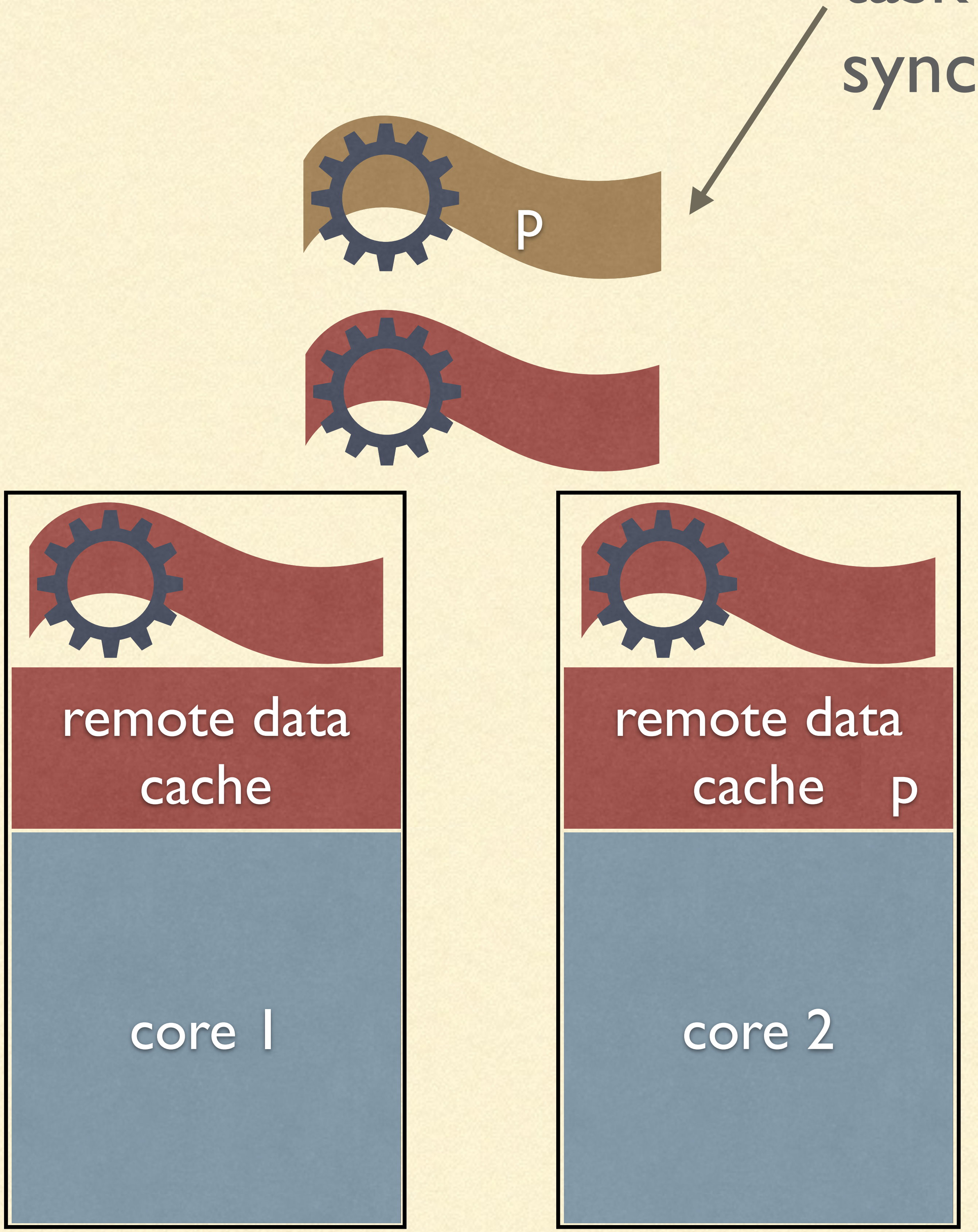
p

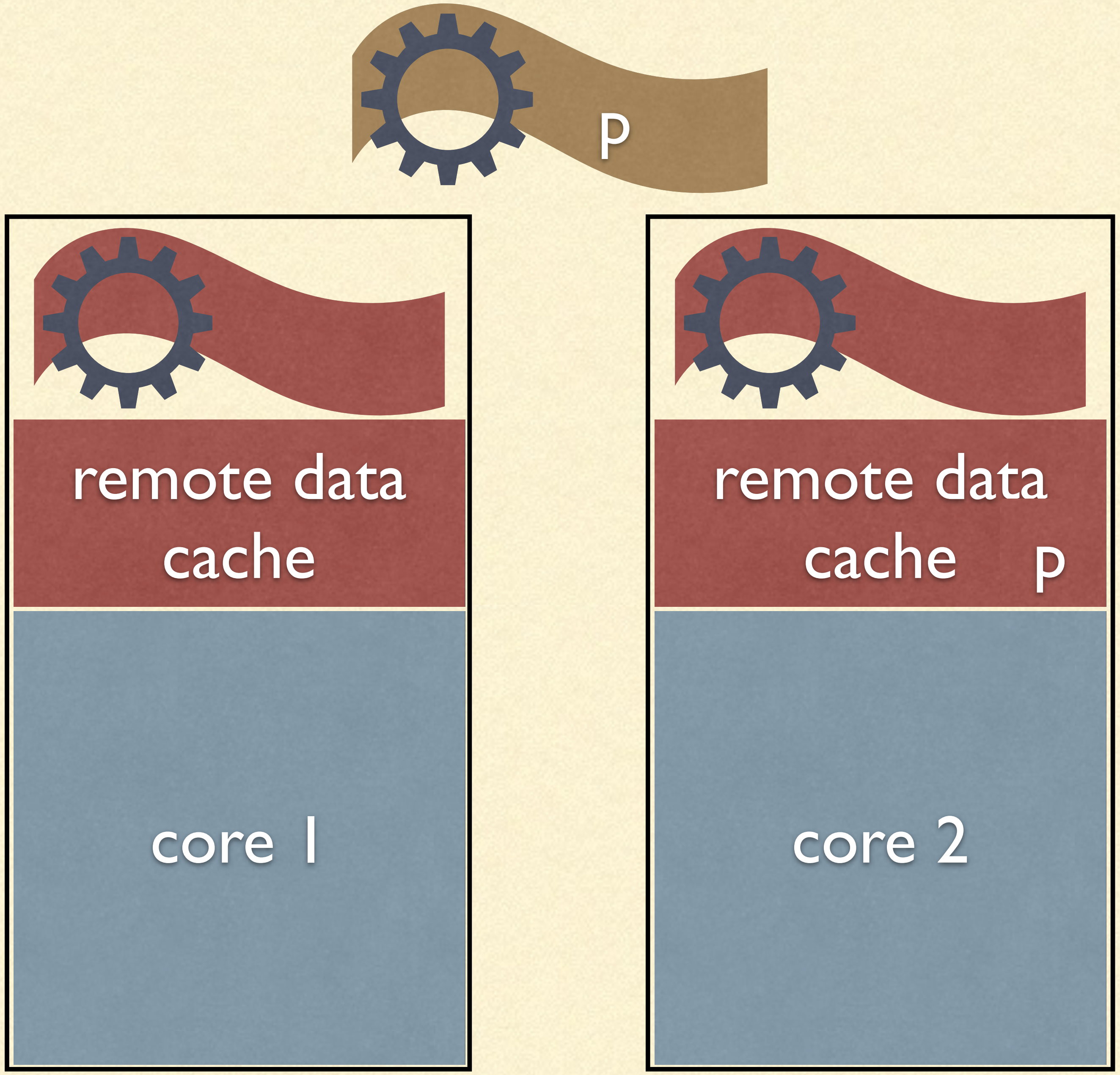
remote data cache

core 1

core 2

task descheduled e.g. in syncvar$.read()





p

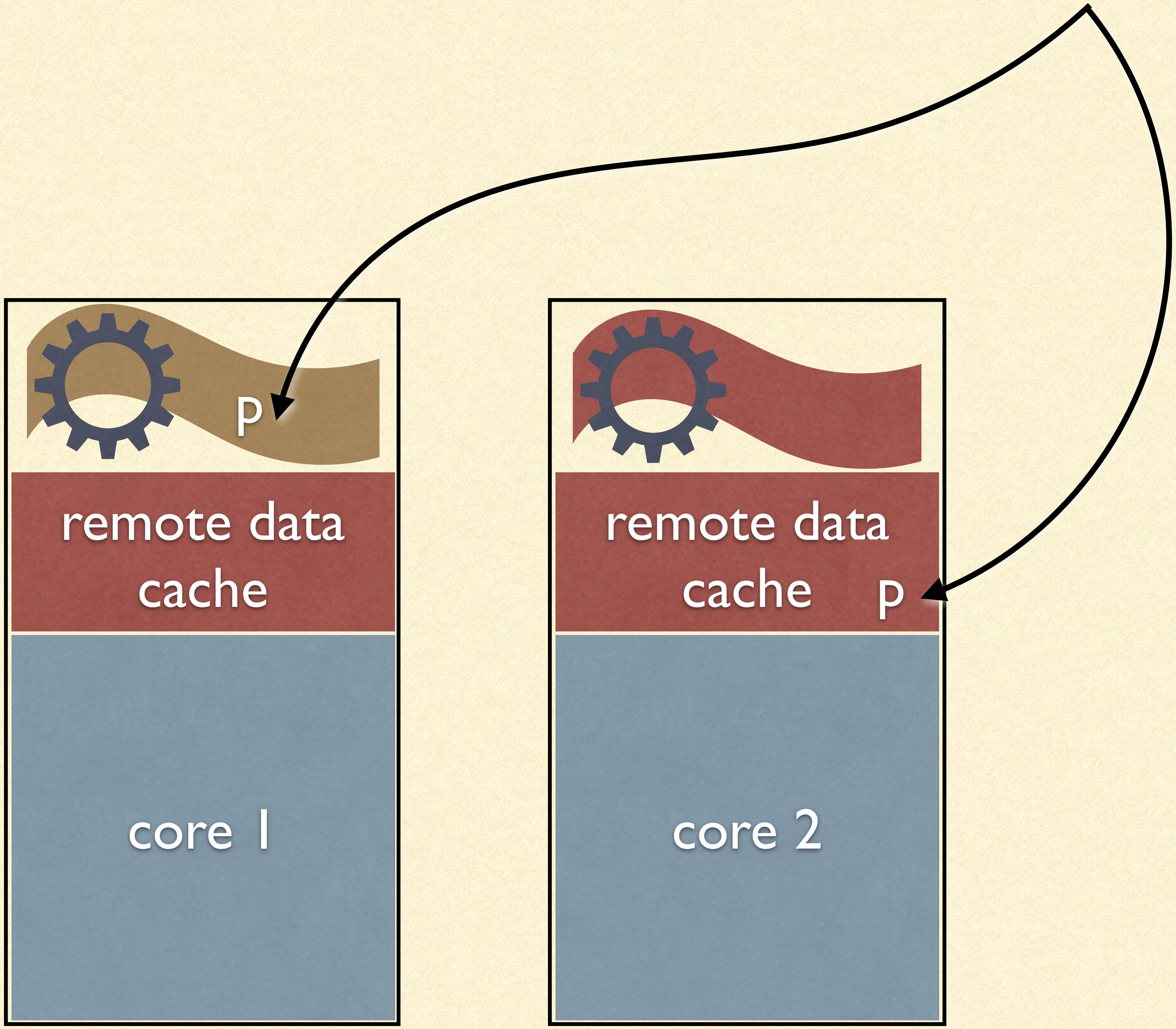
remote data cache

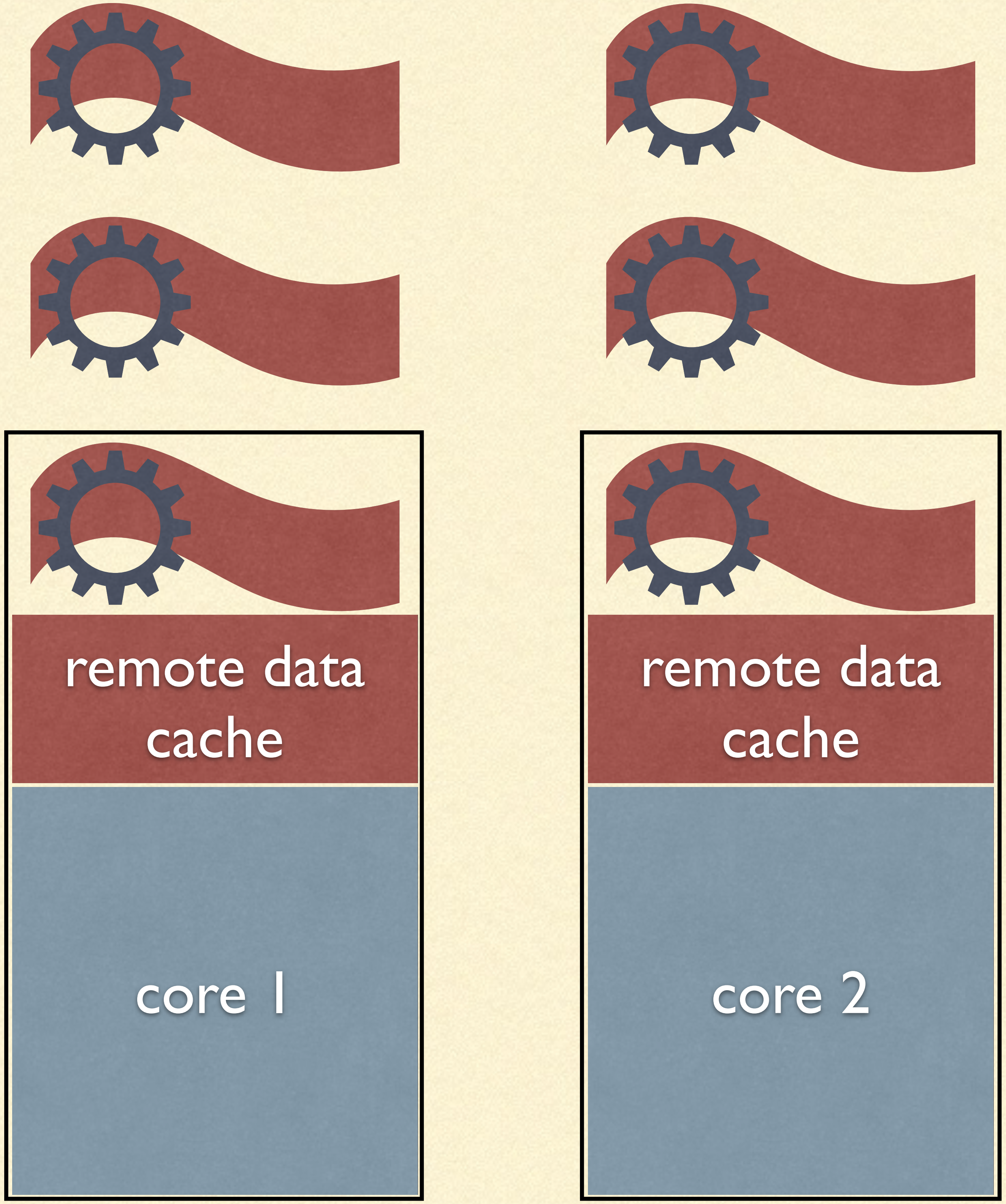
core 1

remote data cache p

core 2

Problem: Operation result is in wrong thread-local storage!





Need Separate Task Queues!

OTHER POSSIBLE SOLUTIONS

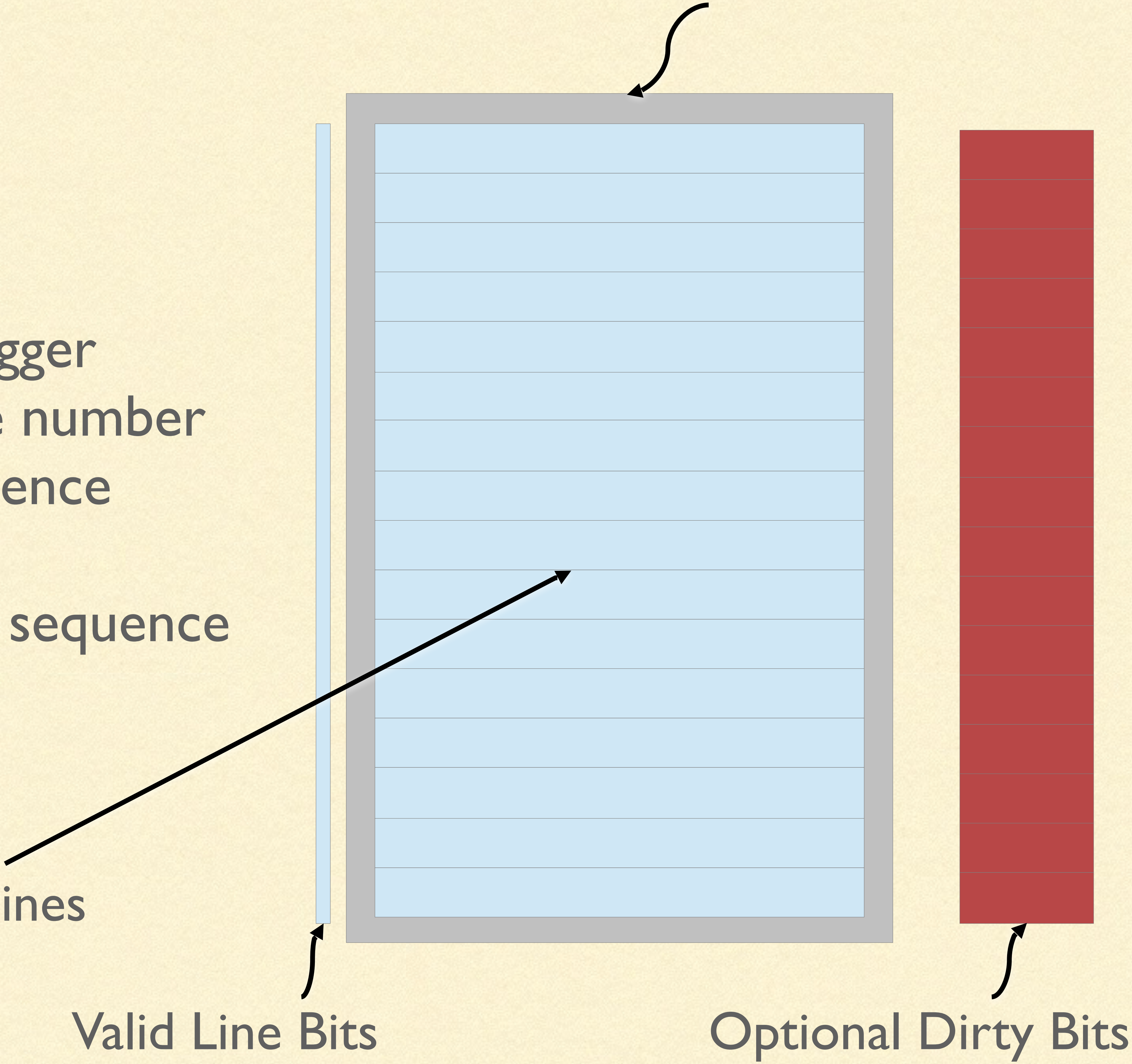
- Pending operations make tasks temporarily un-stealable
- always flush pending operations before descheduling a task and run an *acquire* fence when a task switches threads
- block any descheduled task with pending operations on those operations before it becomes runnable again and run an *acquire* fence when a task switches threads.

LOOKING INSIDE

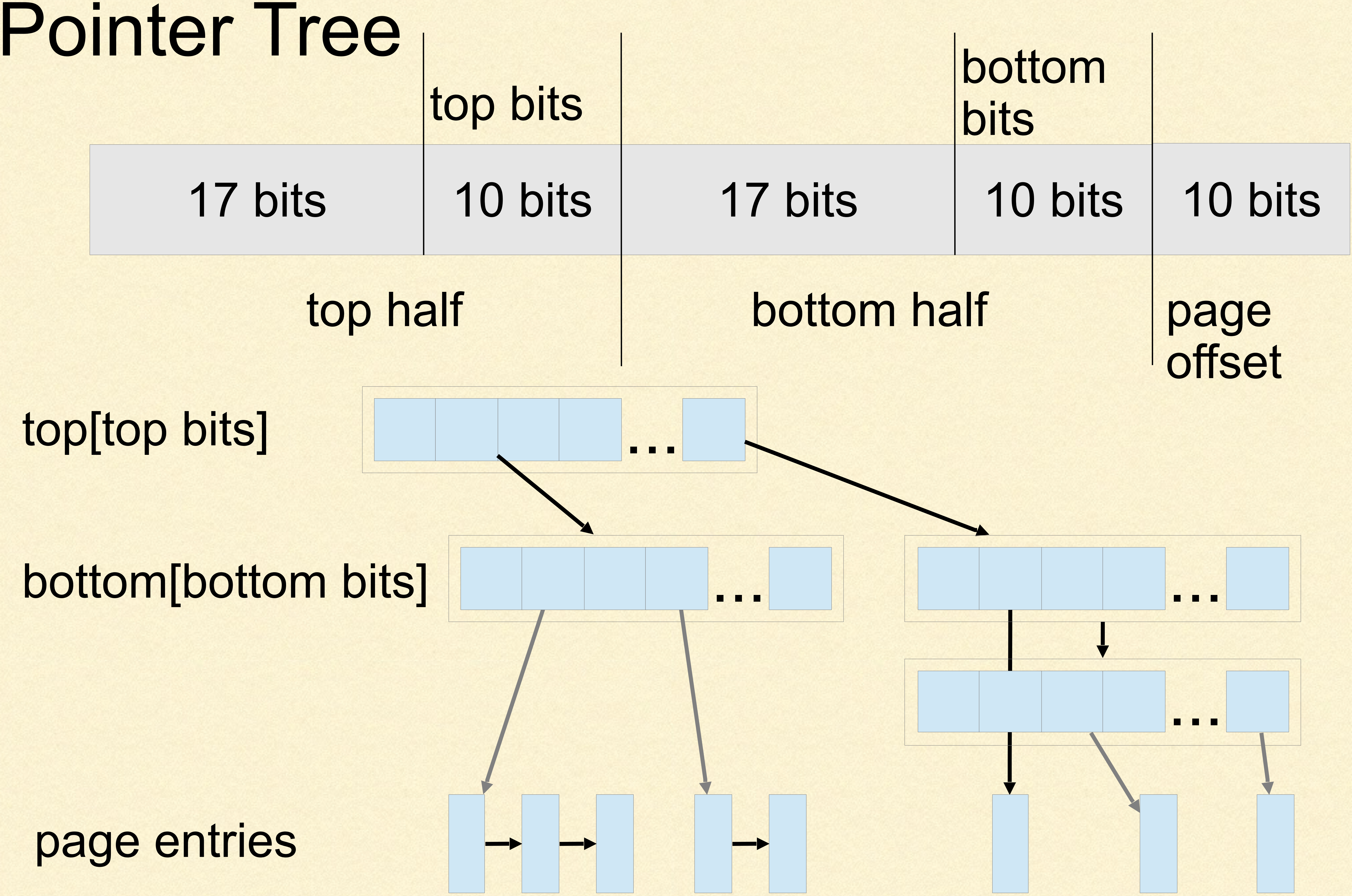
Library of Congress

CACHE ENTRY

- node
- address
- readahead trigger
- min sequence number
- max put sequence number
- max prefetch sequence number

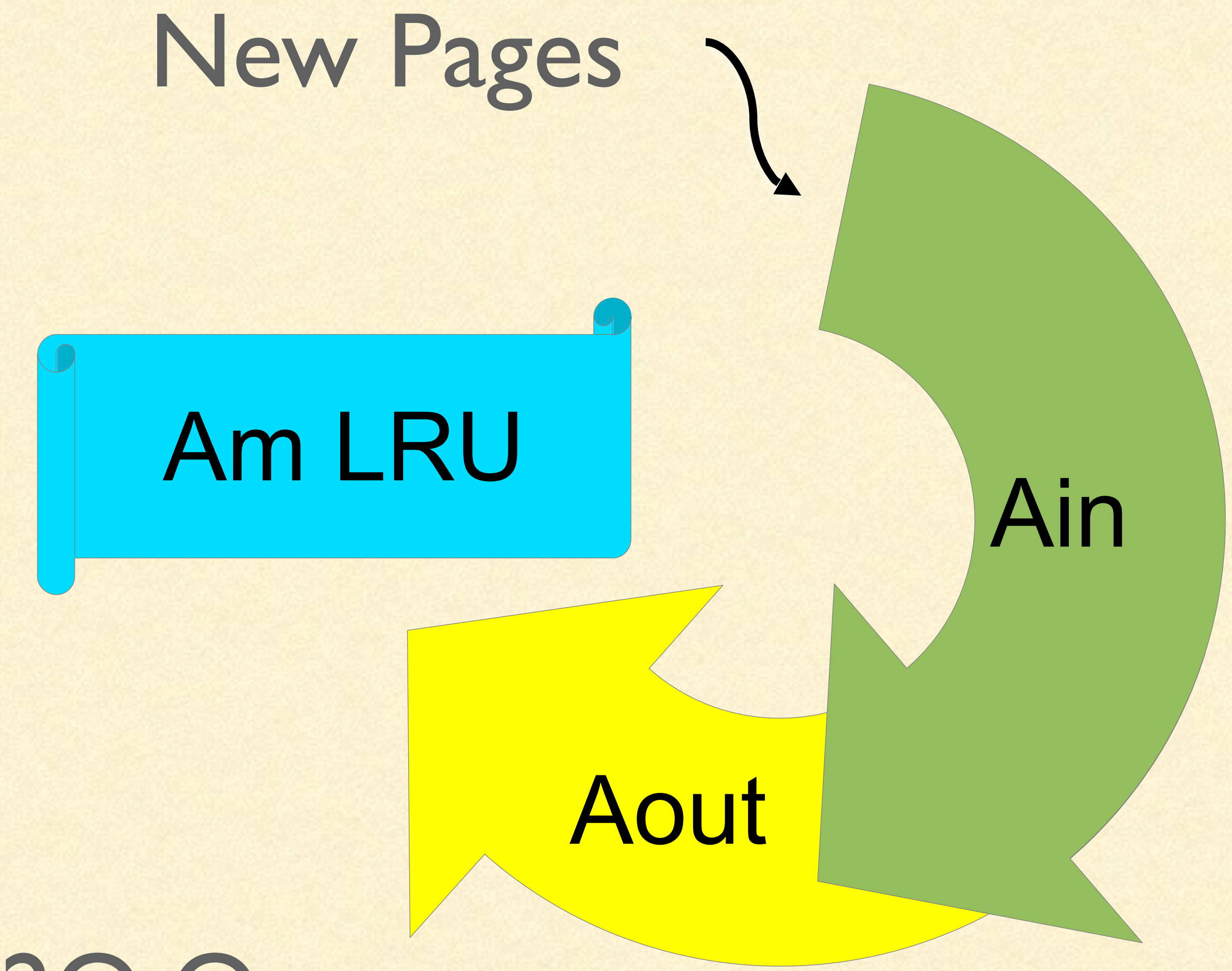


Pointer Tree

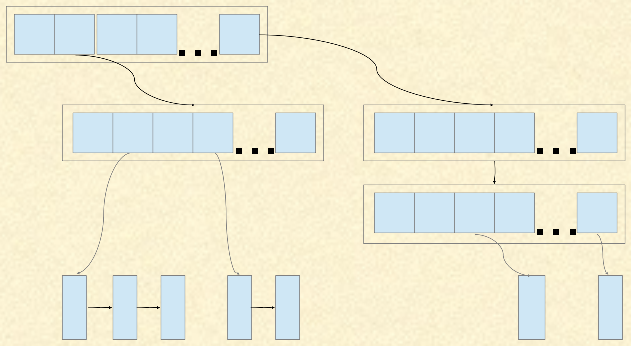


Inspired by “Two Level Tree Structure for Fast Pointer Lookup” by Hans J Boehm

CACHE DATA STRUCTURES



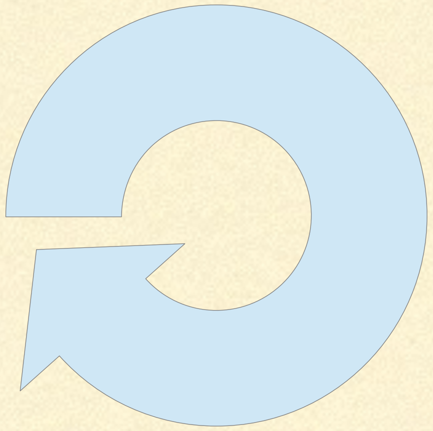
2Q Queues



Pointer Tree

Dirty LRU

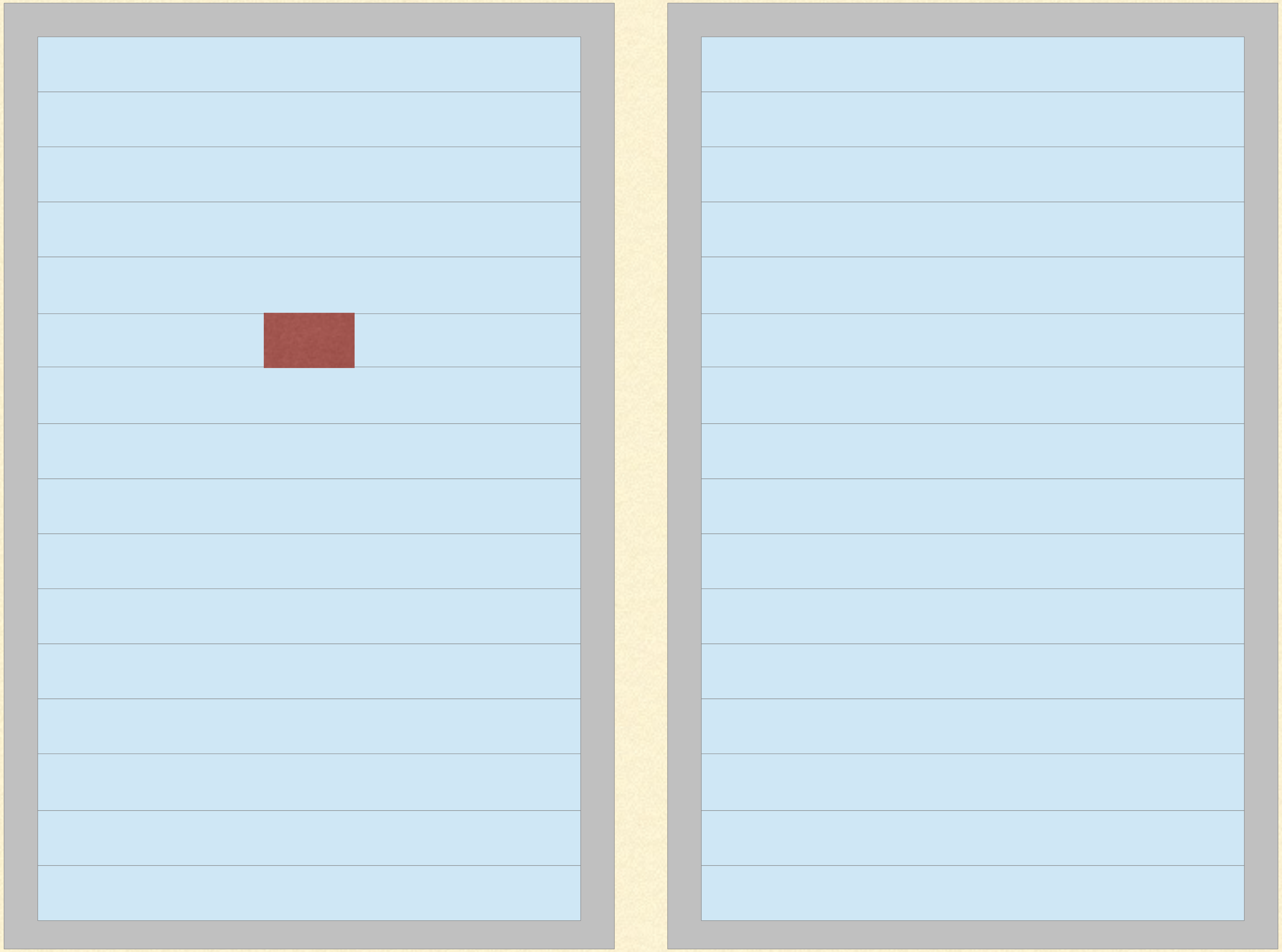
Free Lists



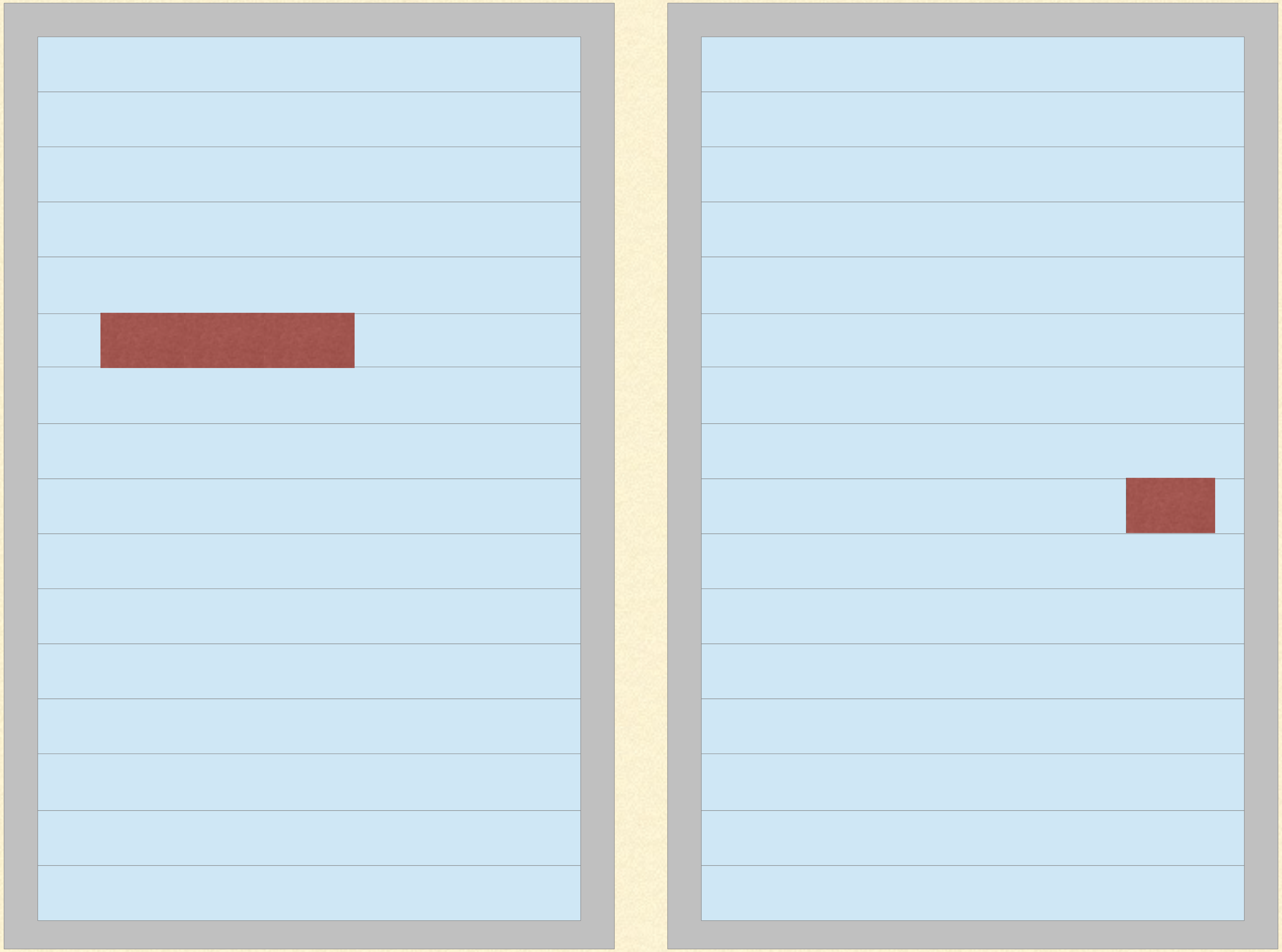
Operations Queue

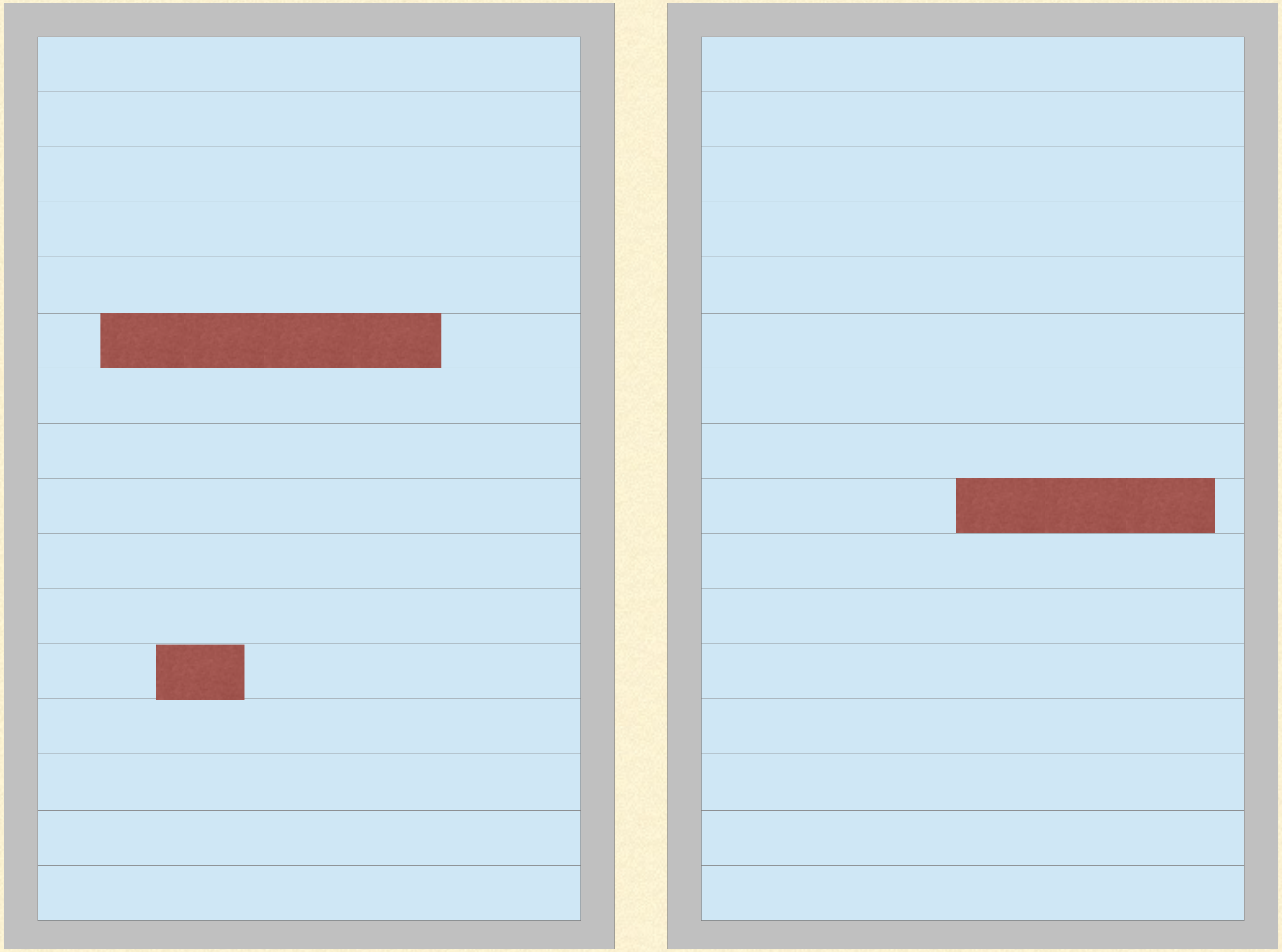
per task:
 last acquire sequence number

WRITE BEHIND



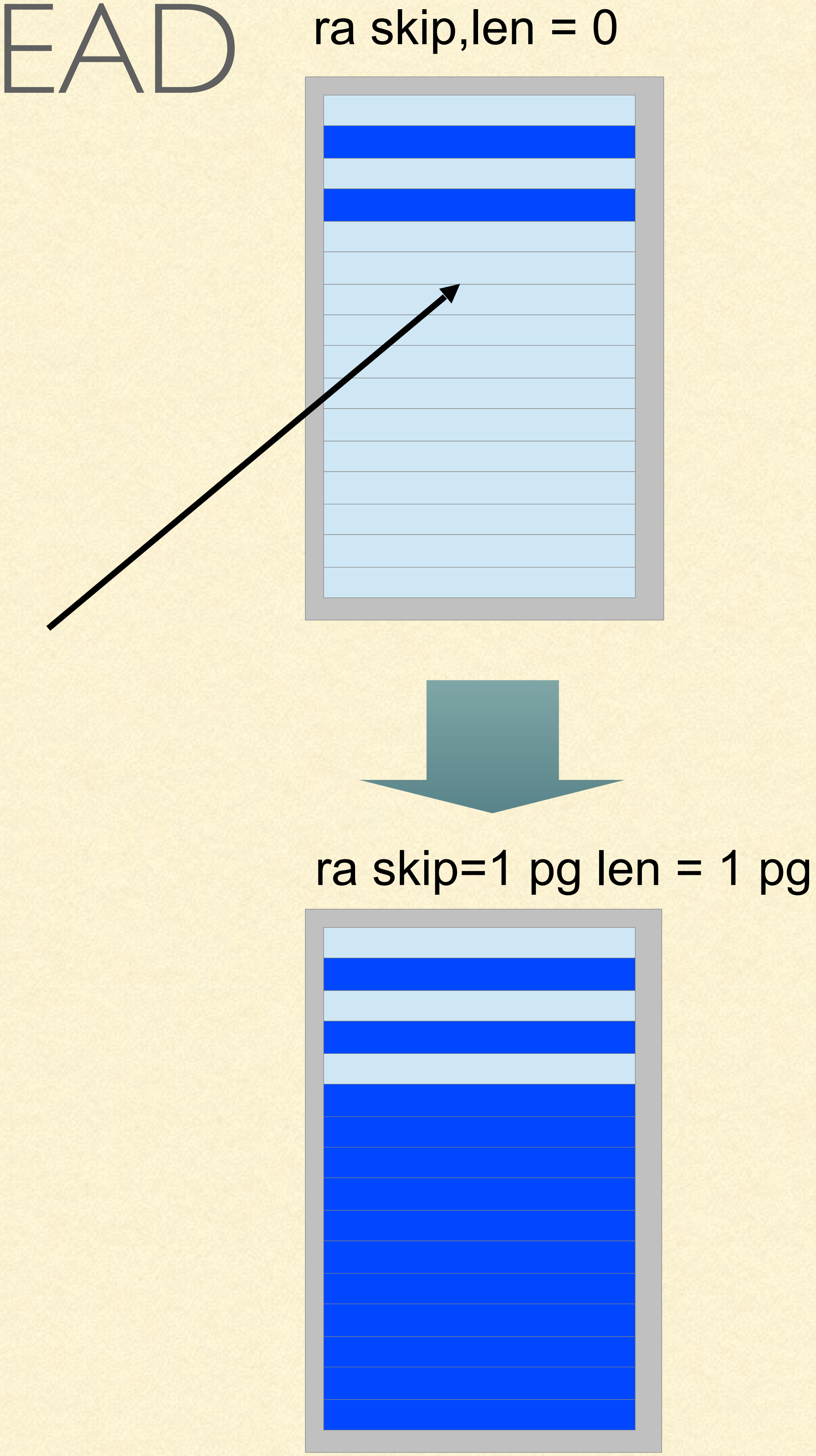
Write Recorded in Dirty Bits, Page added to Dirty Queue





Flushed on *release* or
when there are too many dirty pages

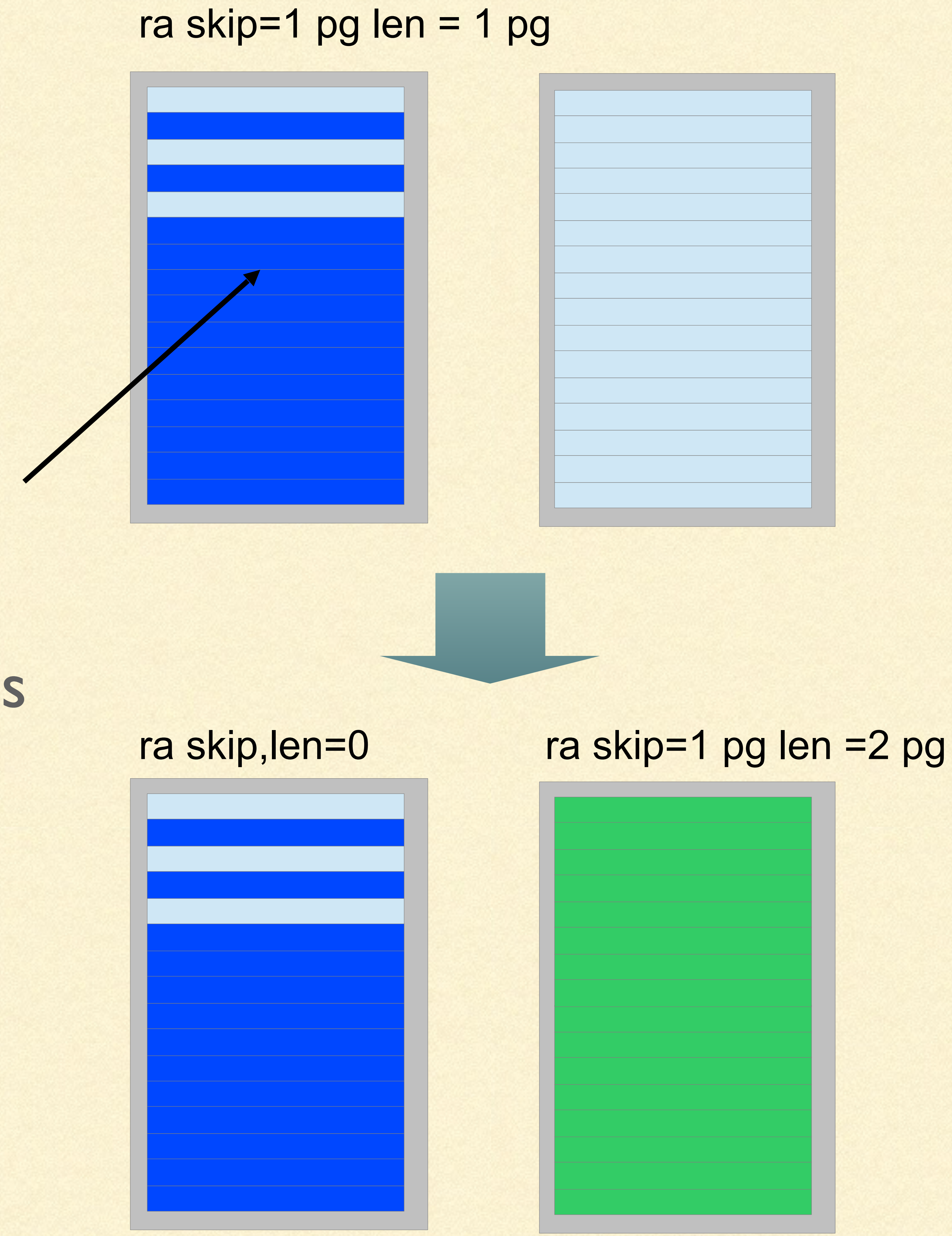
READAHEAD



ra skip,len = 0

ra skip=1 pg len = 1 pg

GET with 2 earlier valid lines triggers synchronous readahead

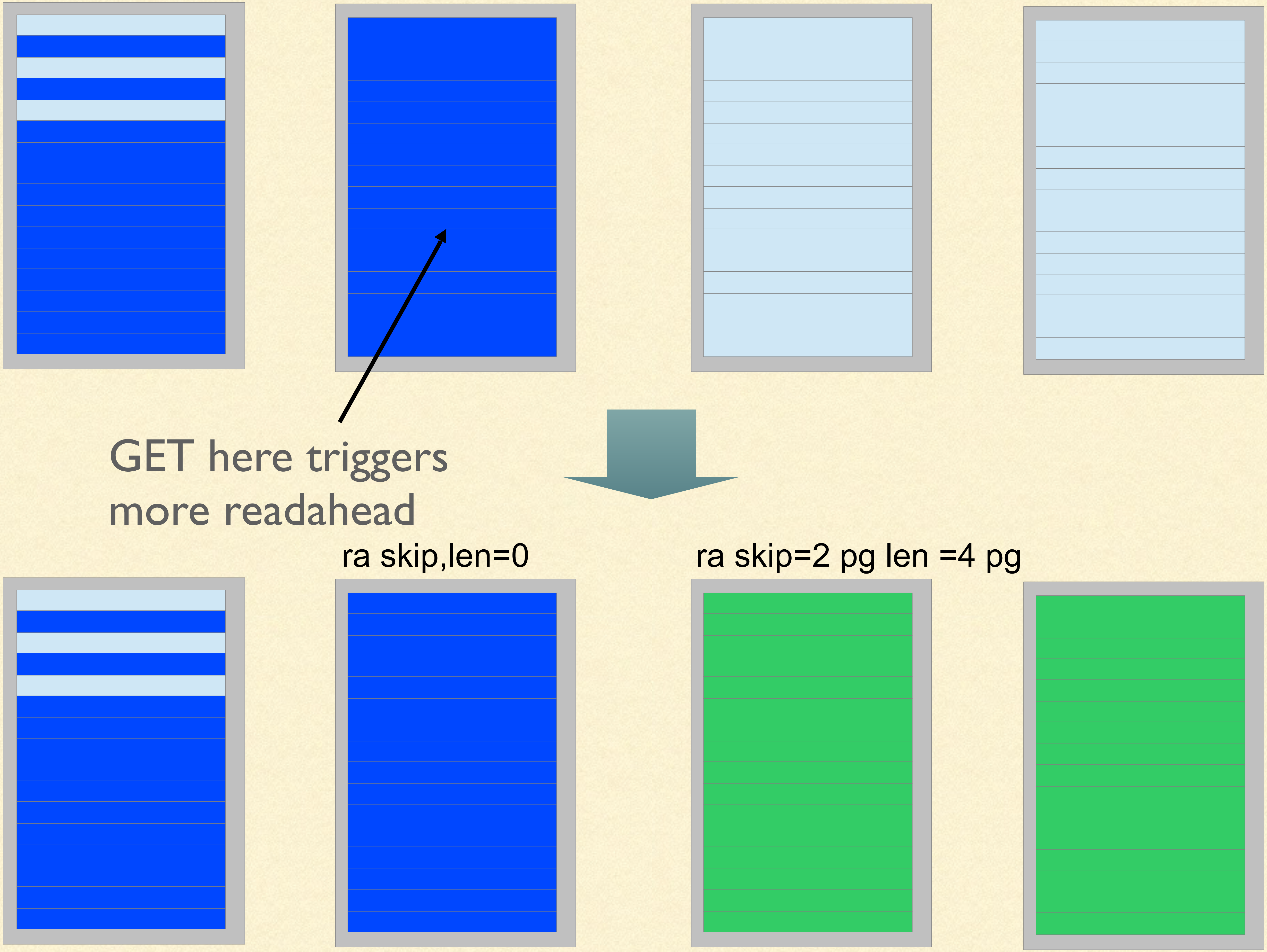


ra skip=1 pg len = 1 pg

ra skip,len=0

ra skip=1 pg len =2 pg

The next GET triggers asynchronous readahead



ra skip,len=0

ra skip=1 pg len =2 pg

GET here triggers more readahead

ra skip,len=0

ra skip=2 pg len =4 pg

Cache for Remote Data:

- is easy to use
- works with naive applications
- shows good benchmark speedups

Library of Congress