Continuous Object Access Profiling and Optimizations to Overcome the Memory Wall and Bloat

Rei Odaira, Toshio Nakatani

IBM Research – Tokyo

ASPLOS 2012 | March 5, 2012

© 2012 IBM Corporation

Many Wasteful Objects Hurt Performance.

- Object-oriented programs allocate many objects. [Zhao et al., 2009]
- Not only many, but also wasteful. [Mitchell et al. 2007]
  - Unused fields, duplicated objects, etc.

→ Called *Memory Bloat*.

☹ Increasing cache misses.

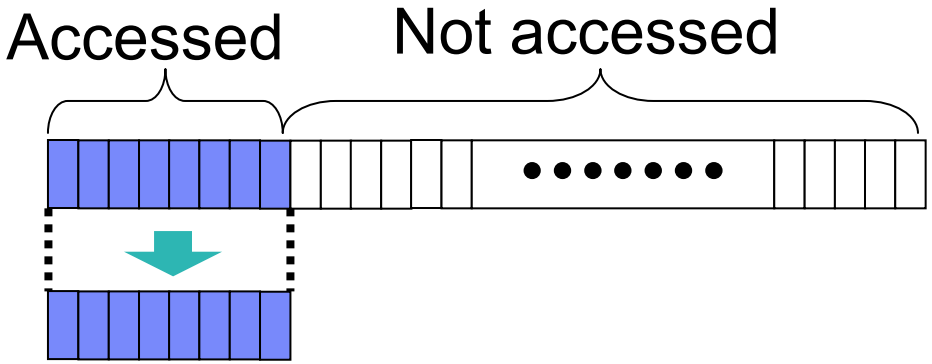
☹ Making the *Memory Wall* higher.

☹ Increasing GC frequency and overhead.

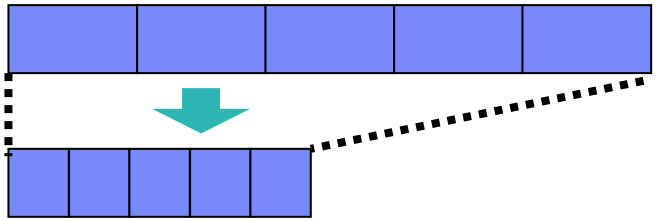
© 2012 IBM Corporation

Object Optimizations to Overcome Memory Bloat

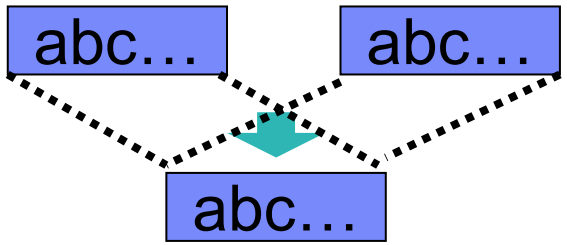
- Allocation size truncation
  [Odaira et al., 2012]



- Object compression
  [Sartor et al., 2008]



- Equal-object merging
  [Marinov et al., 2003]

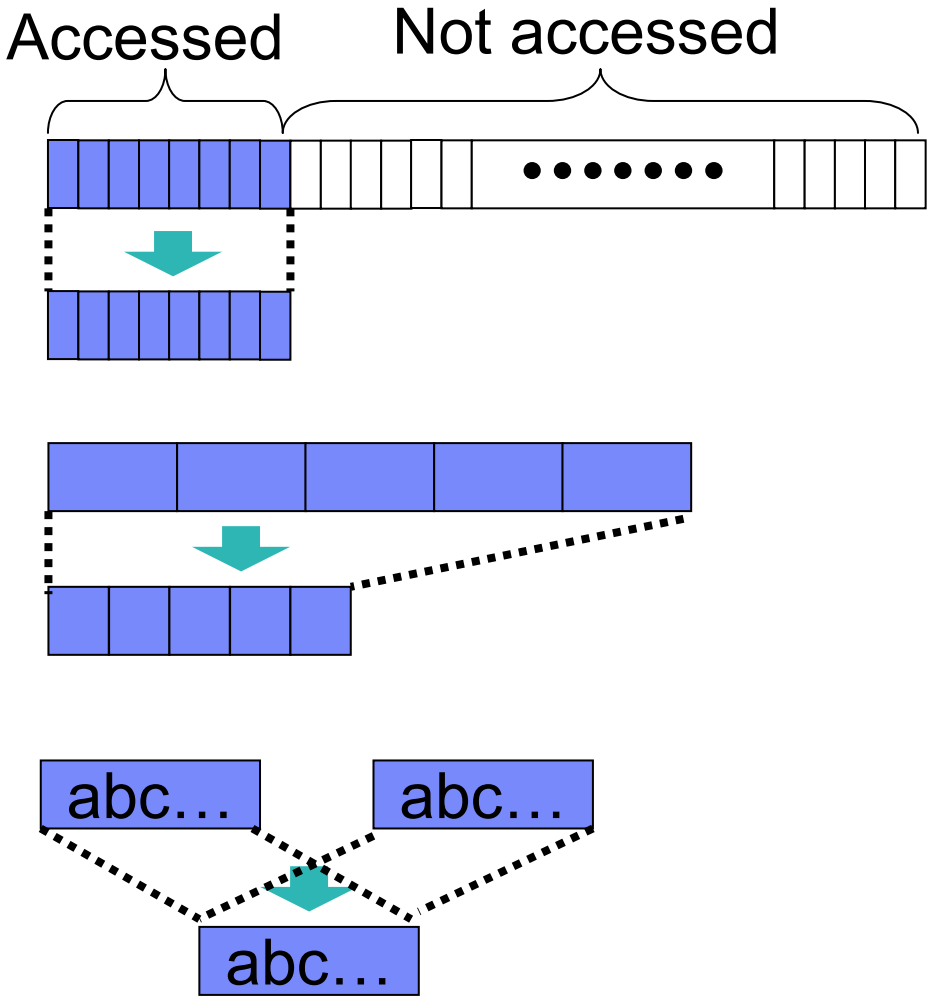


- Lazy allocation, filed reordering, and more.

© 2012 IBM Corporation

Object Optimizations Need Object Access Profiling.

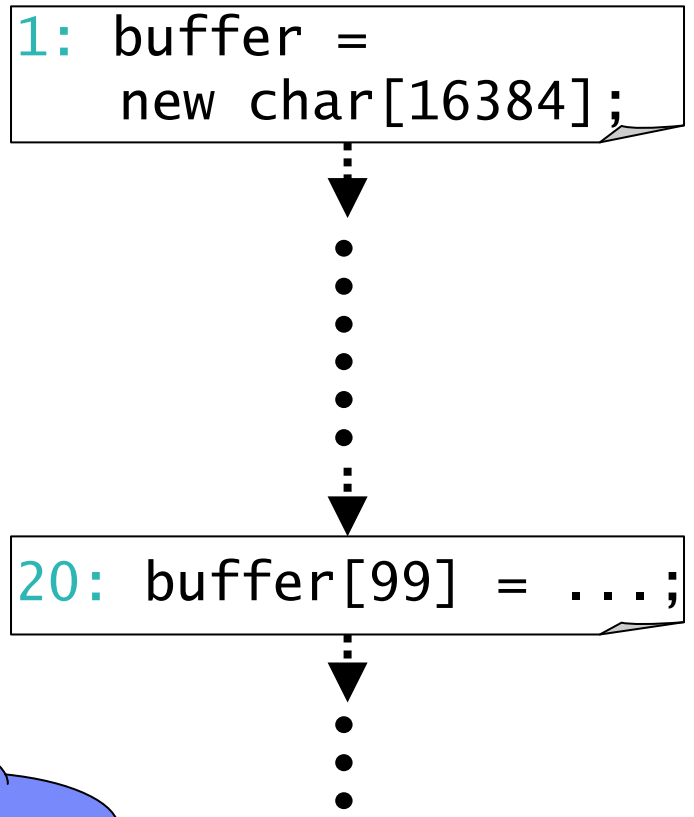
- Allocation size truncation
  - What is the largest accessed index?
- Object compression
  - Are the fields not likely to be accessed?
- Equal-object merging
  - Are the objects equal and likely immutable?



© 2012 IBM Corporation

Definition: Object Access Profiling

- Which instructions access …
- which fields of …
- which objects …
- allocated at which sites
  - … or in which contexts.



Instruction 20 writes to the 99th element of a `char` array allocated at 1.

© 2012 IBM Corporation

Goal: Lightweight Accurate Object Access Profiling

- Lightweight
  - To be used online continuously.
- Accurate
  - Not to miss optimization opportunities.

© 2012 IBM Corporation

Goal: Lightweight Accurate Object Access Profiling

- Lightweight
  - To be used online continuously.
- Accurate
  - Not to miss optimization opportunities.

However, accurate profiling is heavyweight!!

© 2012 IBM Corporation

Goal: Lightweight Accurate Object Access Profiling

- Lightweight
  - To be used online continuously.
- Accurate
  - Not to miss optimization opportunities.

However, accurate profiling is heavyweight!!

- Need to trade off accuracy for low overhead.

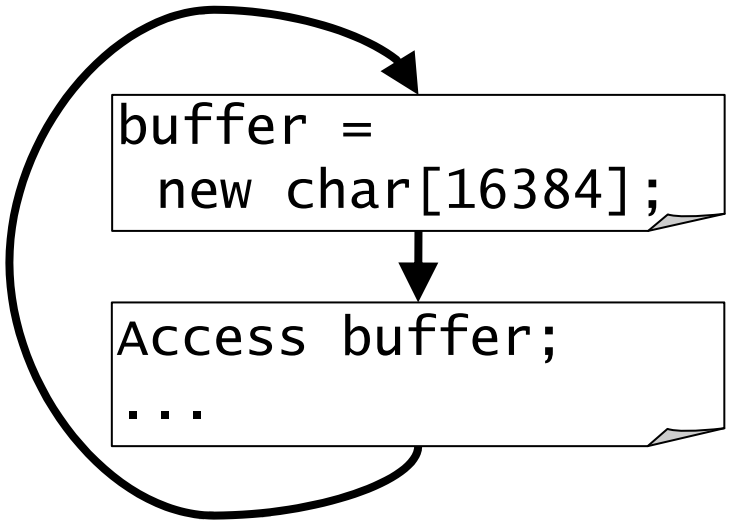
→ What kind of accuracy is really needed?
(… and what can be compromised?)

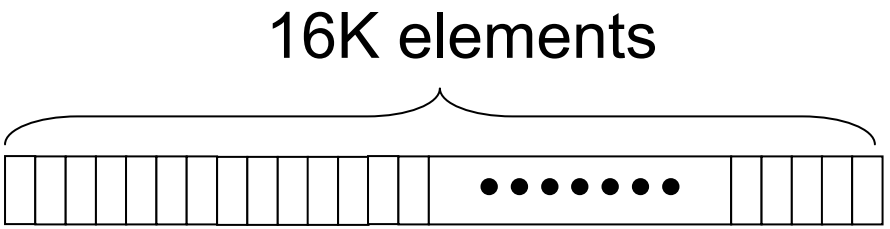
© 2012 IBM Corporation

Example of Memory Bloat: Over-allocated Buffers

- Programmers often allocate many large buffers.
  - E.g. `StringBuffer`, `BufferedReader`, etc. in Java.
- But access only the first few dozen elements.

Lucene search benchmark



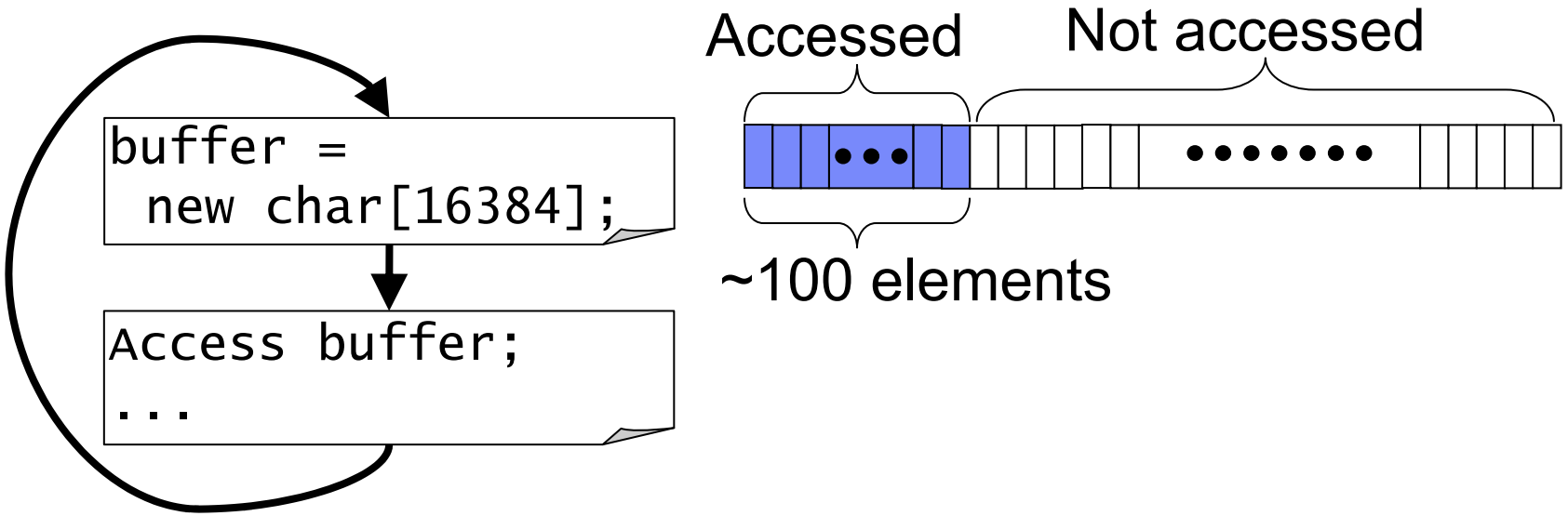


© 2012 IBM Corporation

Example of Memory Bloat: Over-allocated Buffers

- Programmers often allocate many large buffers.
  - E.g. `StringBuffer`, `BufferedReader`, etc. in Java.
- But access only the first few dozen elements.

Lucene search benchmark

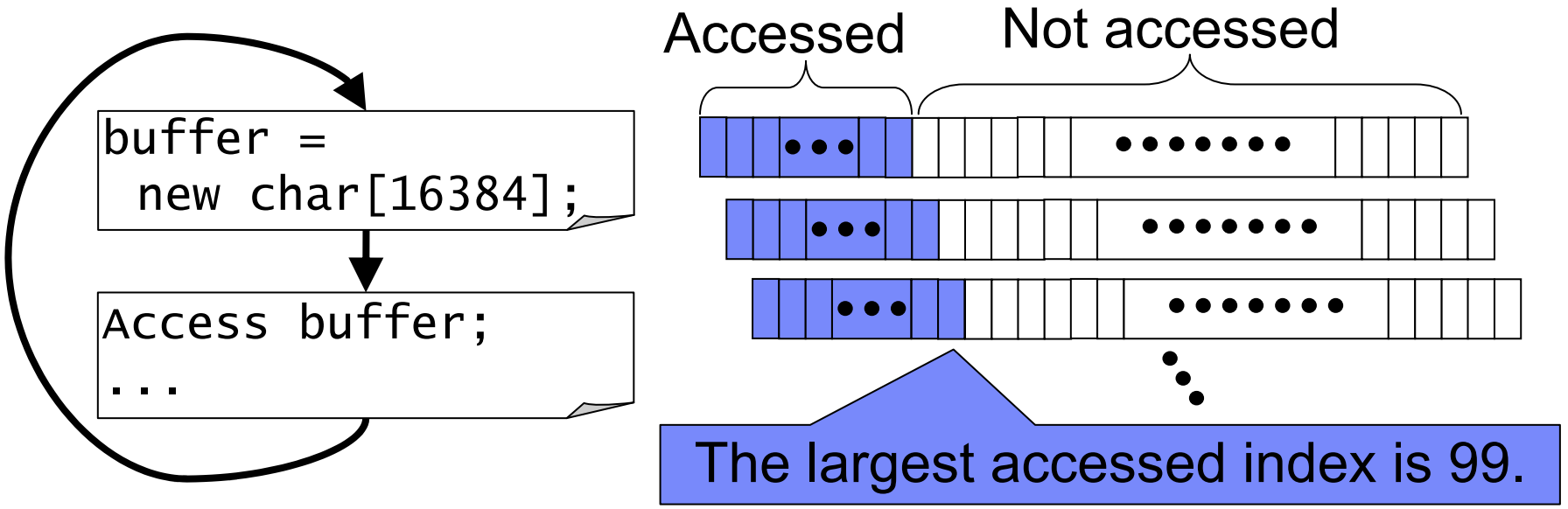


© 2012 IBM Corporation

Example of Memory Bloat: Over-allocated Buffers

- Programmers often allocate many large buffers.
  - E.g. `StringBuffer`, `BufferedReader`, etc. in Java.
- But access only the first few dozen elements.

Lucene search benchmark

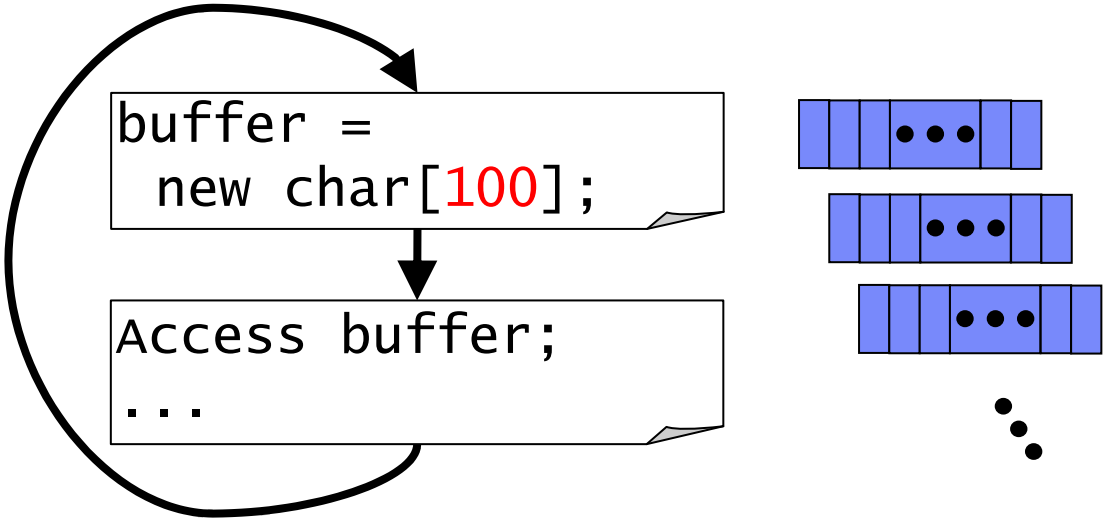


© 2012 IBM Corporation

Example (cont'd): Truncating Allocation Size

- Speculatively allocate smaller buffers.
  - Need a fallback path for speculation failure (See our paper).

Lucene search benchmark

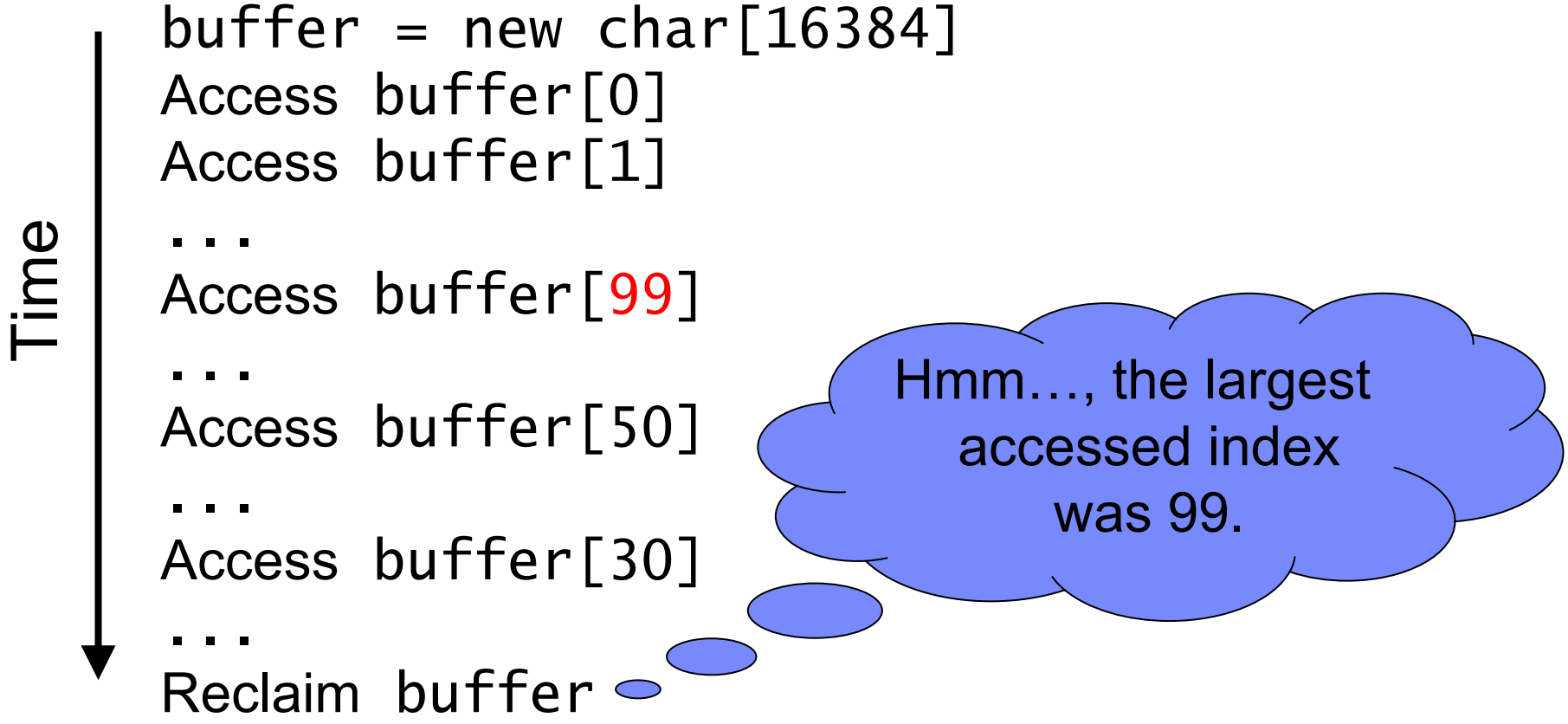


4x speed-up!

© 2012 IBM Corporation

Must Track Object for Its Entire Lifetime.

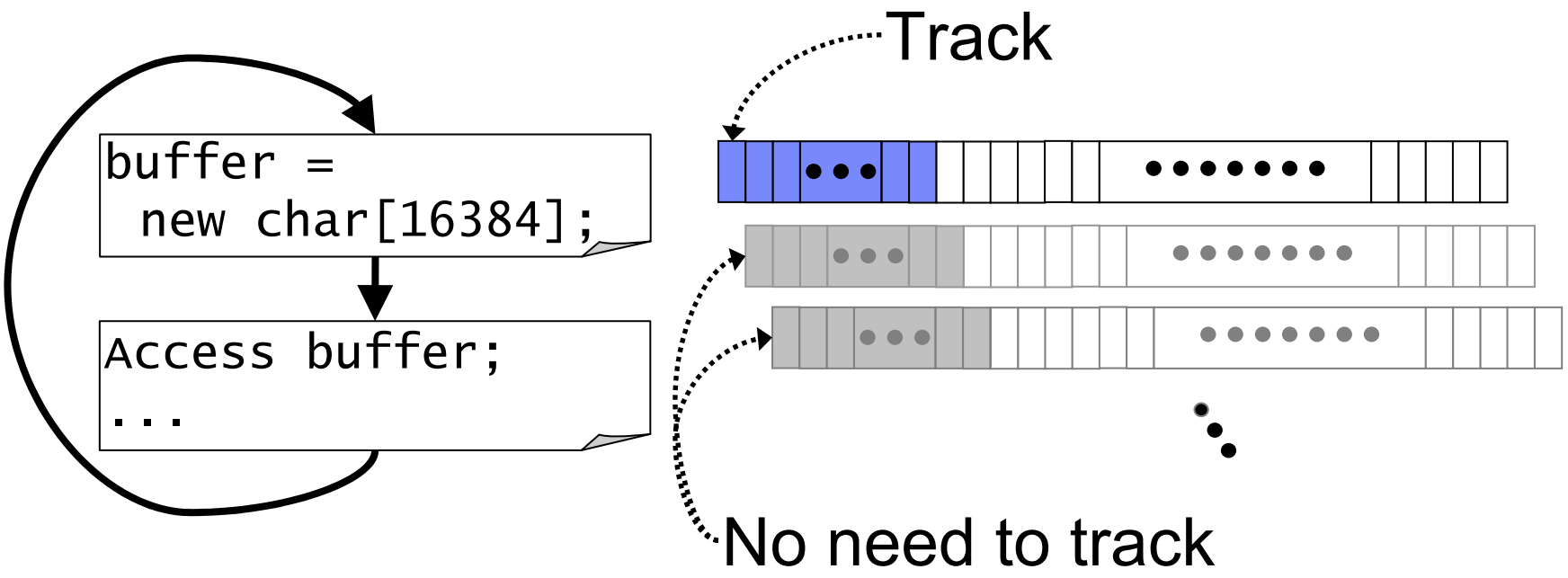
- 99 turns out to be the largest accessed index only after the buffer dies.



© 2012 IBM Corporation

But No Need to Track Every Object.

- Observation:
  Objects allocated at the same site (or context) tend to have the same access pattern.



© 2012 IBM Corporation

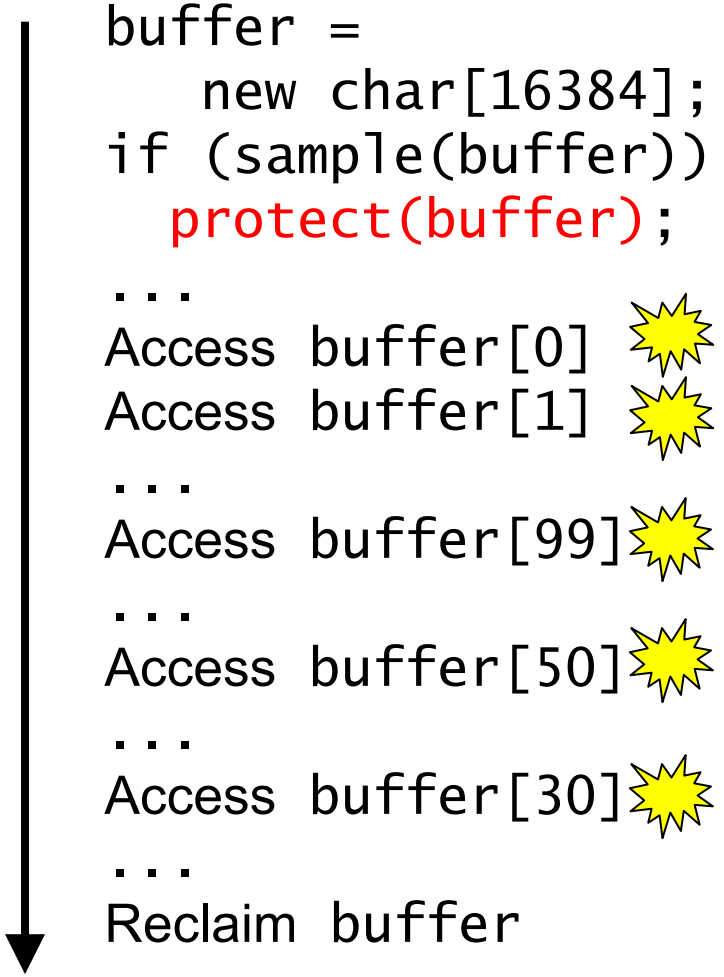
Prior Work is Heavyweight and/or Inaccurate.

- Pointer analysis in Just-In-Time (JIT) compilation
  - ☹ Accurate analysis is heavyweight.
  - ☹ Lightweight analysis is inaccurate (too conservative).
- Code instrumentation
  - ☹ Accurate profiling needs to instrument many accesses.
  - ☹ Lightweight instrumentation cannot track objects' lifetimes.
  - – E.g. Bursty Tracing [Arnold et al., 2001; Hirzel et al., 2001] infrequently samples consecutive accesses.

© 2012 IBM Corporation

Barrier Profiler: Accurate and Lightweight Profiler

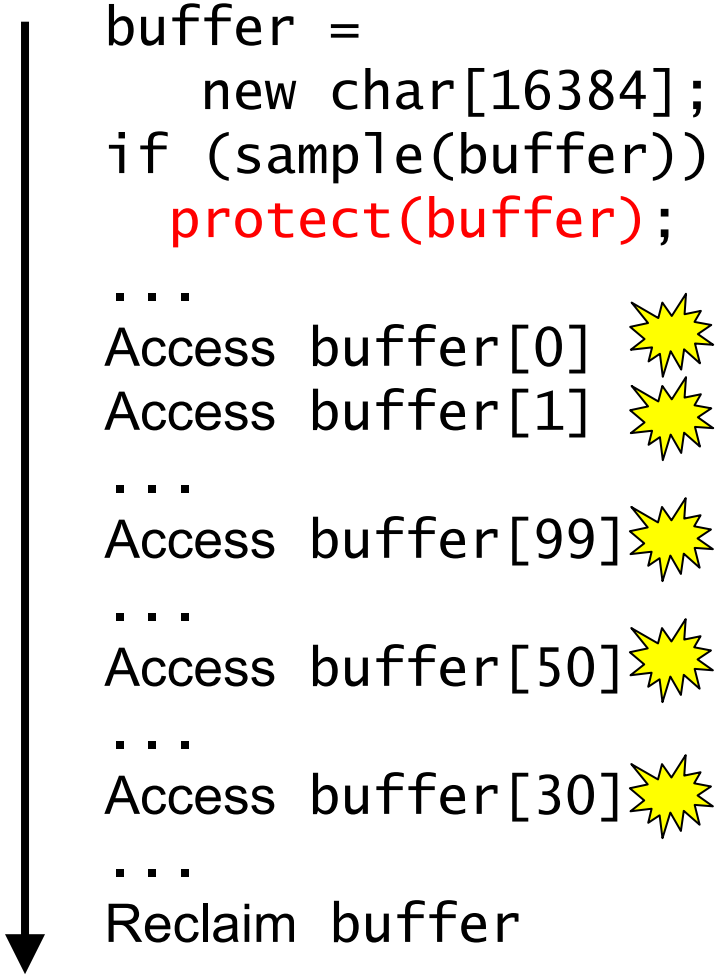
- Memory-protection-based
  - Can track objects' lifetimes.



© 2012 IBM Corporation

Barrier Profiler : Accurate and Lightweight Profiler

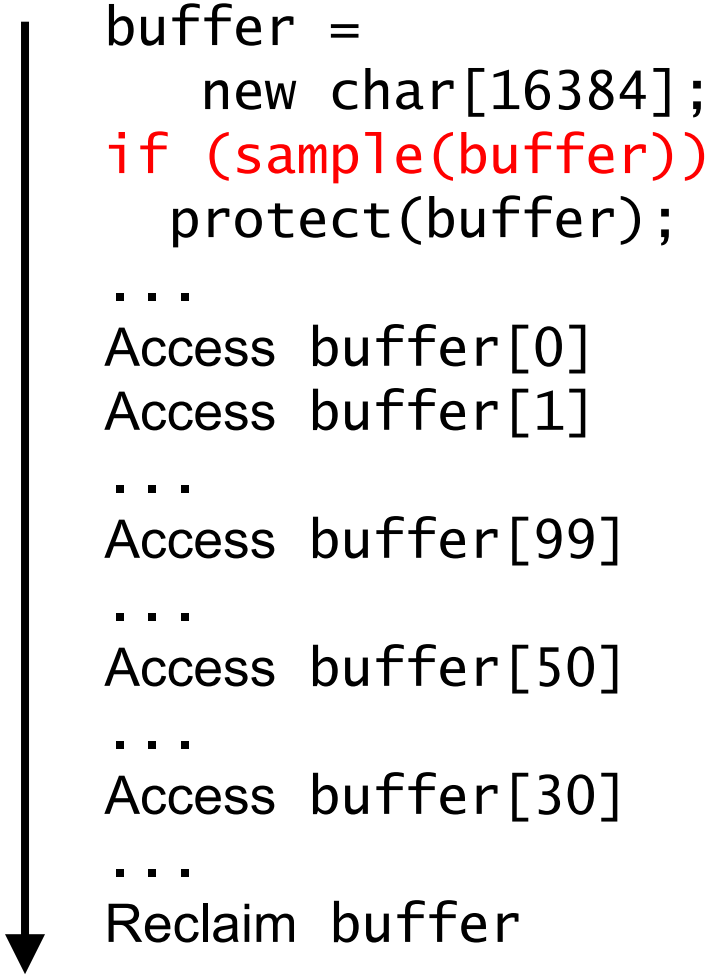
- Memory-protection-based
  - Can track objects' lifetimes.
- Per-object protection
  - *Barrier Pointers*



© 2012 IBM Corporation

Barrier Profiler : Accurate and Lightweight Profiler

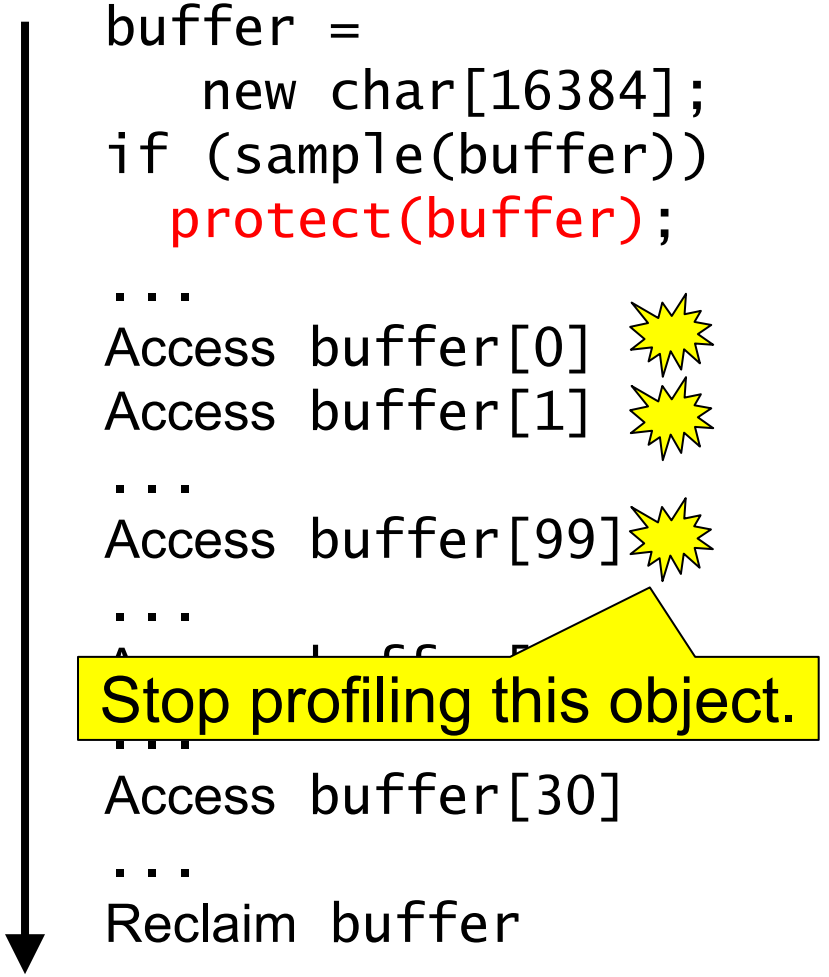
- Memory-protection-based
  - Can track objects' lifetimes.
- Per-object protection
  - *Barrier Pointers*
- Profile-directed overhead reduction
  - Sample objects adaptively.



© 2012 IBM Corporation

Barrier Profiler : Accurate and Lightweight Profiler

- Memory-protection-based
  - Can track objects' lifetimes.
- Per-object protection
  - *Barrier Pointers*
- Profile-directed overhead reduction
  - Sample objects adaptively.
  - Can stop profiling an object.
- ✓ Lightweight
- ✓ Small errors in profiling



© 2012 IBM Corporation

Barrier Profiler : Accurate and Lightweight Profiler

- Memory-protection-based
  - Can track objects' lifetimes.
- Per-object protection
  - *Barrier Pointers*
- Profile-directed overhead reduction
  - Sample objects adaptively.
  - Can stop profiling an object.
- ✓ Lightweight
- ✓ Small errors in profiling

```
buffer =
    new char[16384];
if (sample(buffer))
  protect(buffer);
...
Access buffer[0]
Access buffer[1]
...
Access buffer[99]
...
Access buffer[50]
...
Access buffer[30]
...
Reclaim buffer
```


© 2012 IBM Corporation

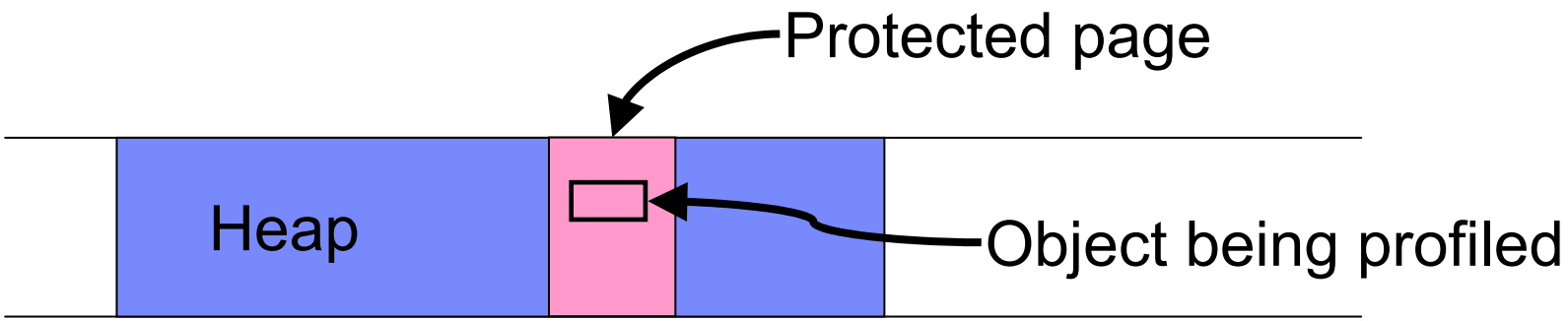
Simple Page Protection Does Not Work.

- How about directly protecting target objects?

☹ Race condition

- Thread A temporarily disables protection to access an object.
- Thread B can access the object without an exception.

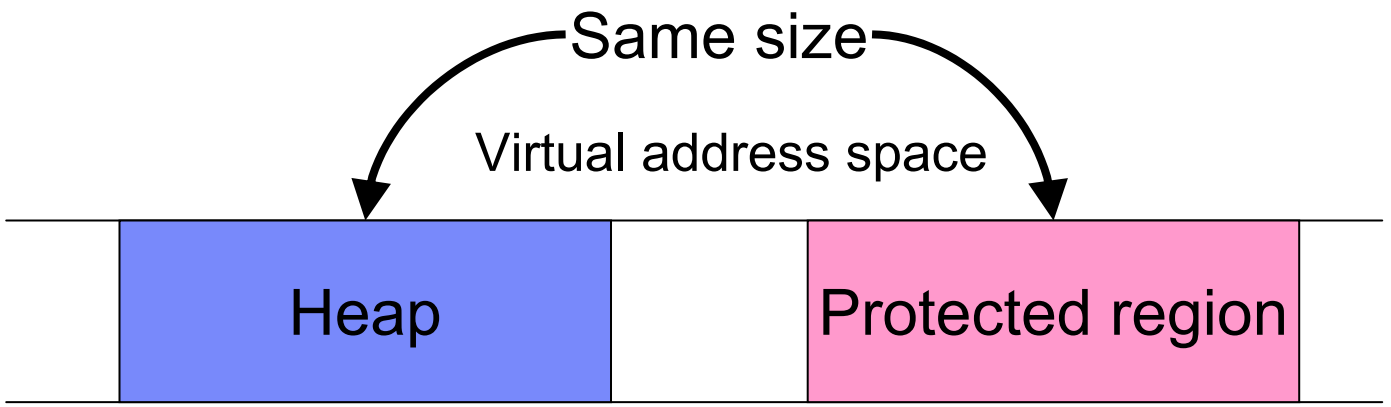
☹ Too coarse-grained



© 2012 IBM Corporation

Barrier Pointers to Enable Per-Object Protection

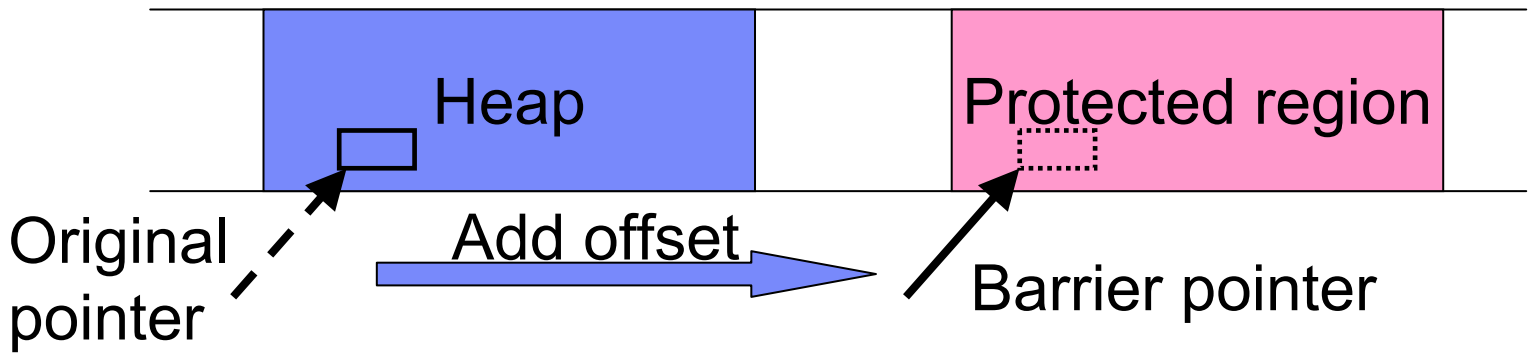
- Reserve a protected region outside of the heap.
  - The same size as the heap, for simplicity.
  - More virtual-memory efficient methods in our paper.
  - No need to assign real memory to the protected region.



© 2012 IBM Corporation

Barrierization

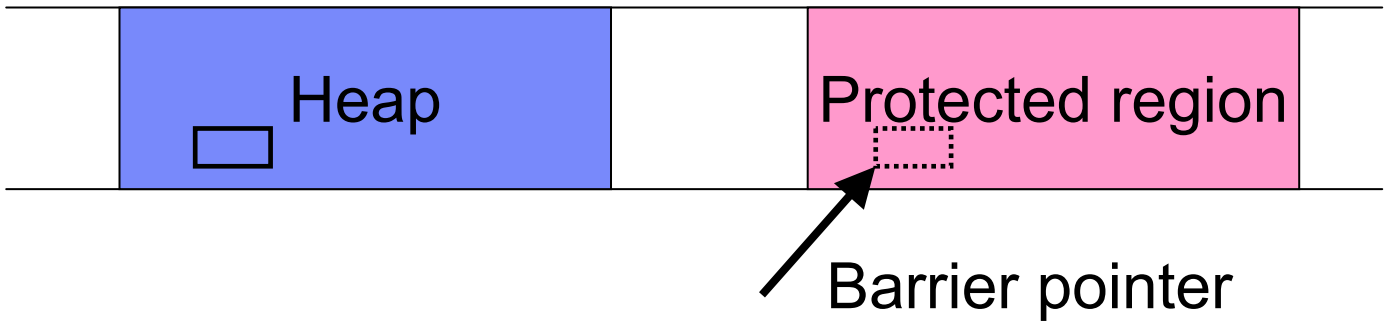
- Convert all pointers to a target object to *barrier pointers*.
  - Add the constant offset.
- Barrier pointers point to the protected region.
- Done at object allocation time.



© 2012 IBM Corporation

Barrier Pointers Enable Per-Object Profiling.

- Accesses via the barrier pointers cause exceptions.
  - Profiling in the exception handler.
- ☺ Accesses to the other objects cause no exception.
- Subtle issues explained in our paper:
  - Handling pointers to the middle of an object.
  - Executing atomic memory accesses.



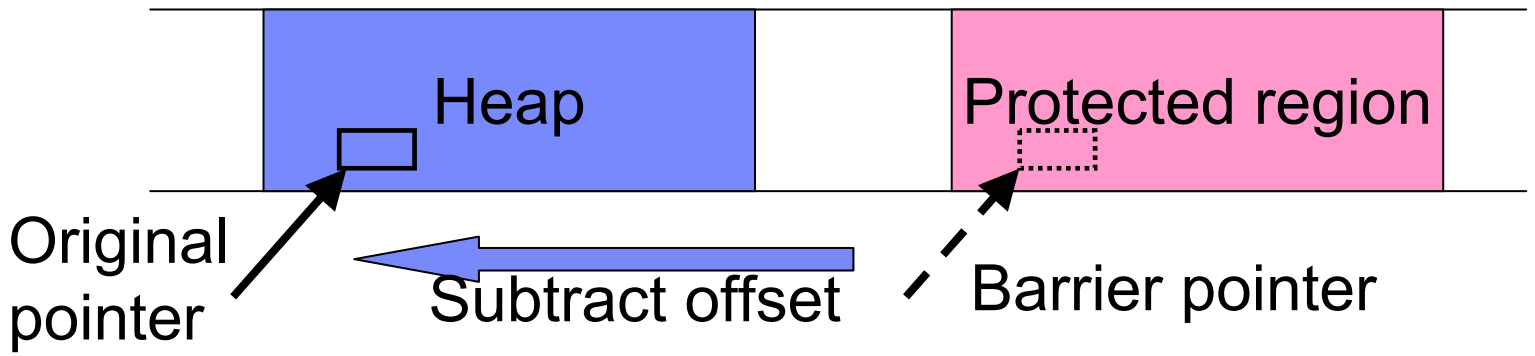
© 2012 IBM Corporation

Unbarrierization

- Restore the original pointer from the barrier pointer.
  - Subtract the constant offset.

☺ Can access the object without disabling protection.

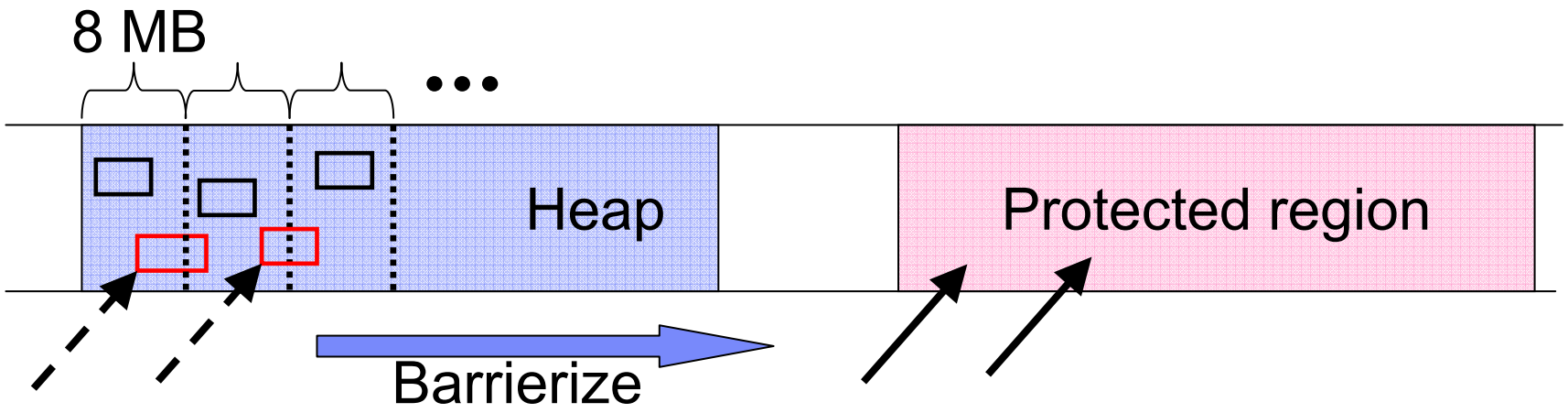
☺ No race condition



© 2012 IBM Corporation

Barrier Profiler Samples Objects at Allocation Time.

- Per *n*-MB allocation.
  - n = 8, by default, but it is adaptive.



© 2012 IBM Corporation

Experiments

- Implemented in 32-bit IBM J9/TR 1.6.0 SR6.
- Metrics
  - Accuracy
    → Smaller errors in profiles than Bursty Tracing
    (Details in our paper)
  - Performance overhead
- Simulated Bursty Tracing
(access-sampling-based code instrumentation).
  - Accuracy estimated using offline perfect tracing.
  - Overhead calculated based on full-instrumentation overhead and sampling ratio (200:1).

© 2012 IBM Corporation

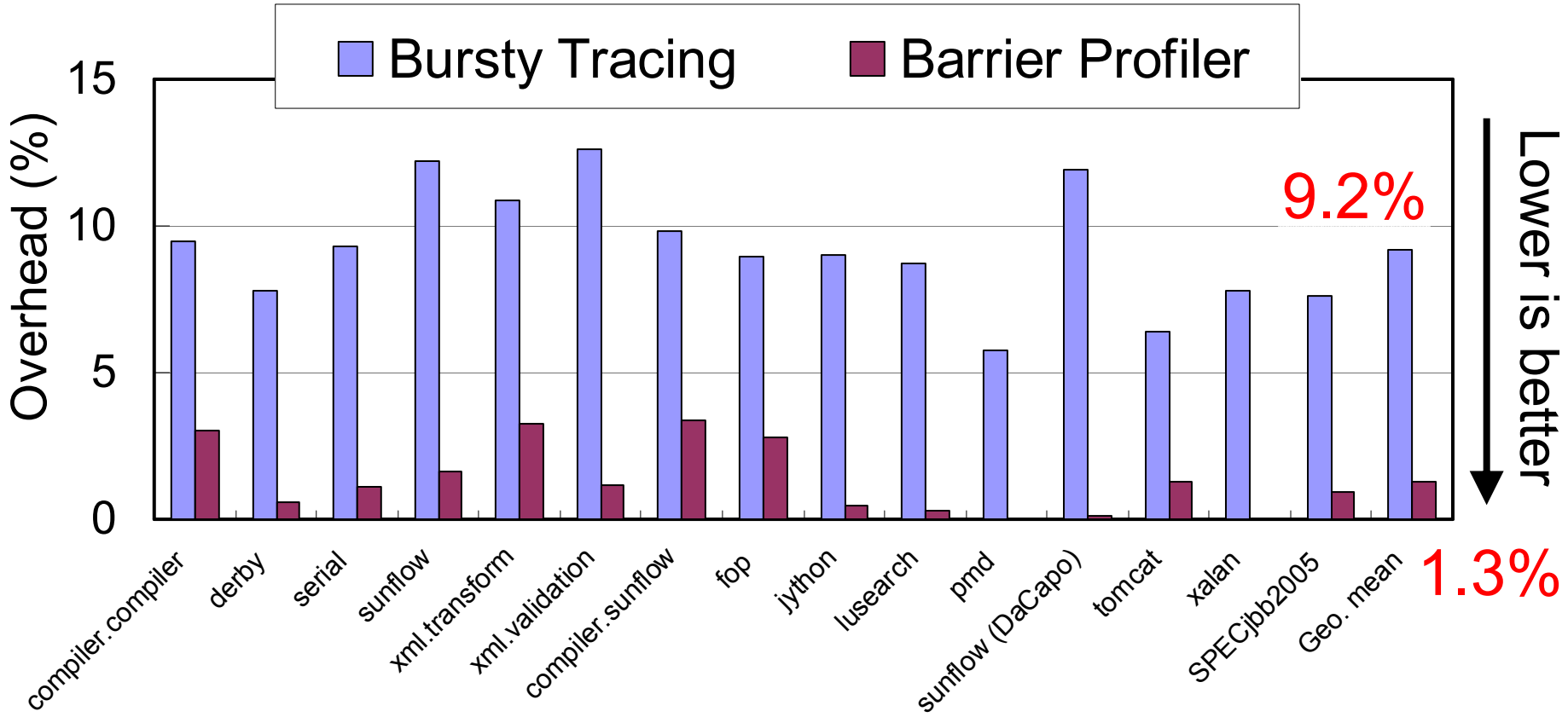
Experimental Environment and Benchmarks

- Environment
  - 4-core 2-SMT 4.7-GHz POWER6, 32-GB main memory
  - Linux 2.6.18, 1 GB Java heap
- Benchmarks
  - SPECjvm2008, DaCapo 9.12, and SPECjbb2005.
  - Only allocation-intensive programs shown in this talk.

© 2012 IBM Corporation

Performance Overhead

- Bursty Tracing: 9.2%, Barrier Profiler 1.3%
  - Without the profile-directed overhead reduction, the overhead of Barrier Profiler was 75.2%.



© 2012 IBM Corporation

Online Object Optimizations Using Barrier Profiler

- Online compression of character arrays
  - 8.6% speed-up in SPECjbb2005.
- Online truncation of allocation sizes
  - 36% speed-up in “lusearch”, with 1 GB Java heap
  - 4x speed-up in “lusearch”, 6% speed-ups in “xalan”, with 2x minimum Java heap.

© 2012 IBM Corporation

Conclusion

- Barrier Profiler:
  accurate and lightweight object access profiler
  - Smaller errors in profiles than Bursty Tracing
  - 1.3% performance overhead on average

✓Indispensable to continuous online profiling.

- Online object optimizations using Barrier Profiler
  - Online truncation of allocation sizes
  - Online compression of character arrays

✓Enabled for the first time by Barrier Profiler.

© 2012 IBM Corporation

Thank you!

- Questions?

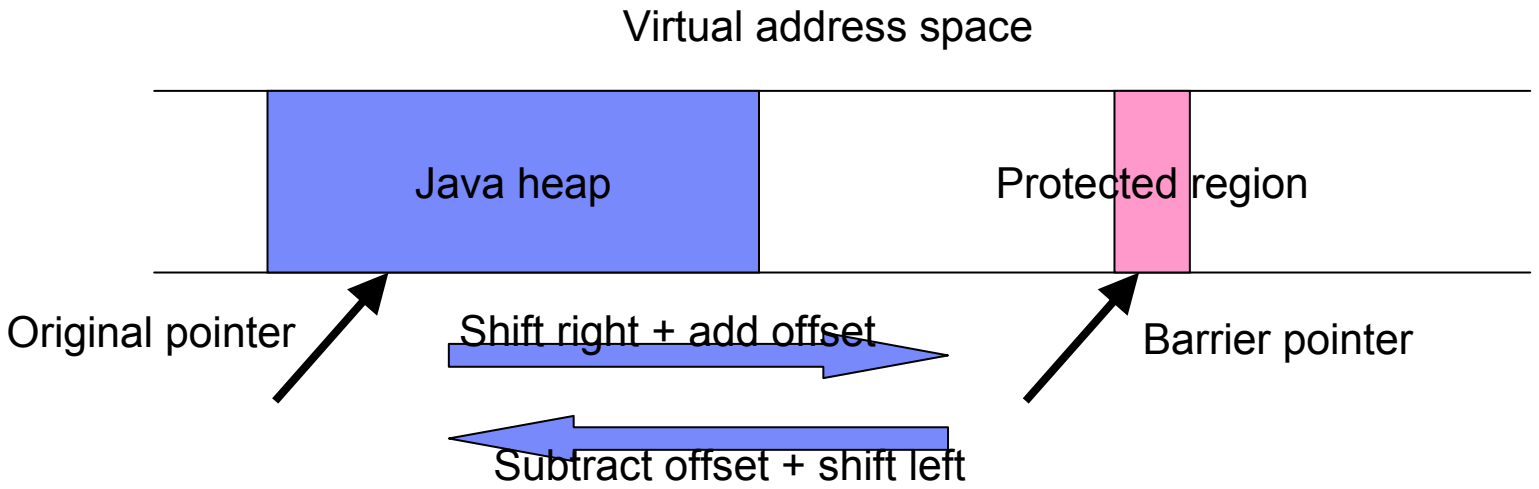
© 2012 IBM Corporation

Backup

© 2012 IBM Corporation

Advanced Barrier Pointers

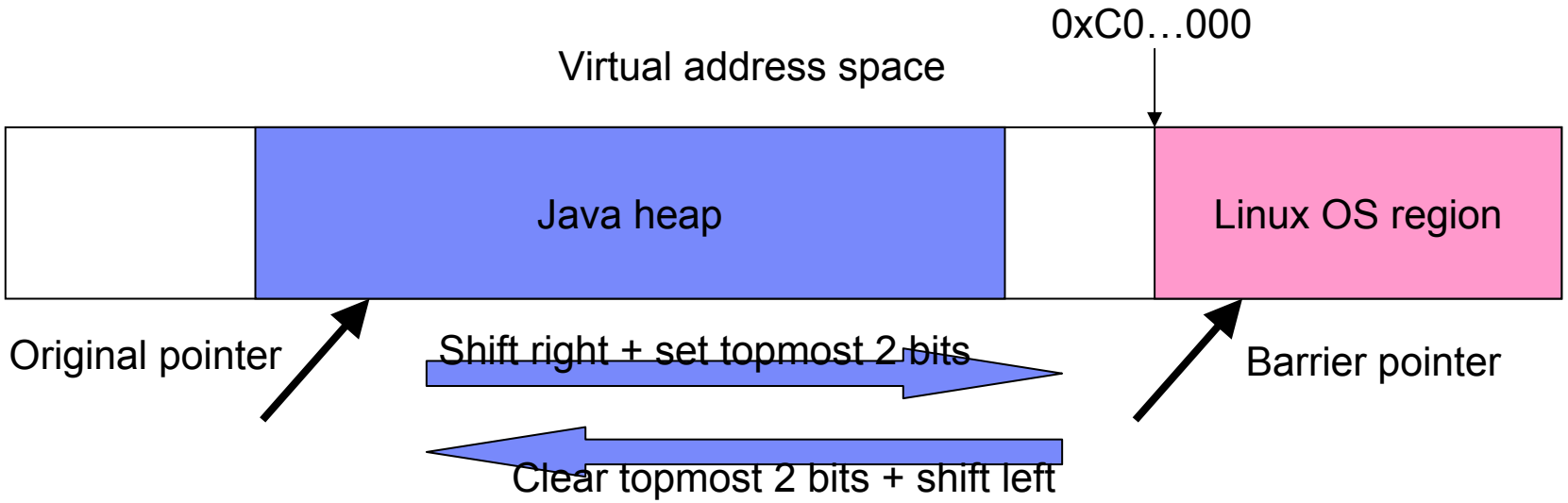
- Shift right/left for barrierization/unbarrierization.
  - Objects are 8-byte aligned.
- Protected region can be 1/8 of the Java heap.



© 2012 IBM Corporation

Utilizing OS-protected Region

- Linux occupies the highest 1/4 of a virtual space.
  - Inaccessible from users.
- Set/clear the topmost 2 bits for barrierization/unbarrierization.



© 2012 IBM Corporation

How to Evaluate Accuracy

- Profile all of the accesses to all of the allocated objects. (= Full trace)
  - Using the barrier pointer framework.
- Calculate the percentages of allocated bytes satisfying the following properties, at each allocation site.
  - Write-only objects
  - Immutable objects
  - Non-accessed bytes
- Compare the profilers' results with the full trace, using the absolute differences of the calculated percentages.

- Simulated Bursty Tracing: estimate the accuracy by sampling 10,000 memory accesses after skipping 2,000,000 accesses in the full trace.

© 2012 IBM Corporation

Online Adjustment of Allocation Sizes

- Focus on “growable array” programming pattern.
  - E.g. `StringBuffer` in Java.
  - Speculation failure check is already coded.

- Programmers use a new API to wrap the target allocation sites.
- Feedback the best size to each allocation site.

© 2012 IBM Corporation

New API

```
   public final class System {
1: public static char[] getCharArrayOfBestSize(int defaultSize) {
2:   return new char[defaultSize];
3: }
   ...
   }
   public class BufferedReader {
4: public BufferedReader(Reader in) {
5:   this.in = in;
6:   //this.cb = new char[8192];  // Original implementation
7:   this.cb = System.getCharArrayOfBestSize(8192);
8:   this.length = this.cb.length;
9: }
   ...
   }
```


© 2012 IBM Corporation

Overhead Reduction Exploits *Boringness.*

- *Interesting properties*:
  - Properties of objects which object optimizations exploit.
  - E.g. Write-only, immutable, non-accessed, etc.
- *Boring objects*:
  - Objects that no longer satisfy interesting properties.
  - E.g. objects once read are no-longer write-only.

→ Stop profiling of such objects by unbarrierization.

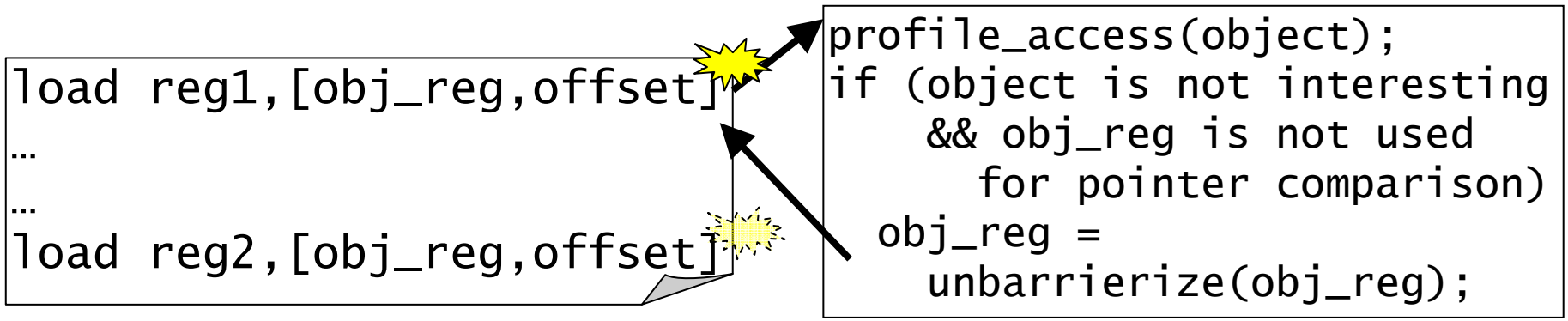
- *Boring allocation sites*:
  - Sites that have allocated many boring objects.

→ Reduce sampling frequency at such sites.

© 2012 IBM Corporation

Techniques for Overhead Reduction

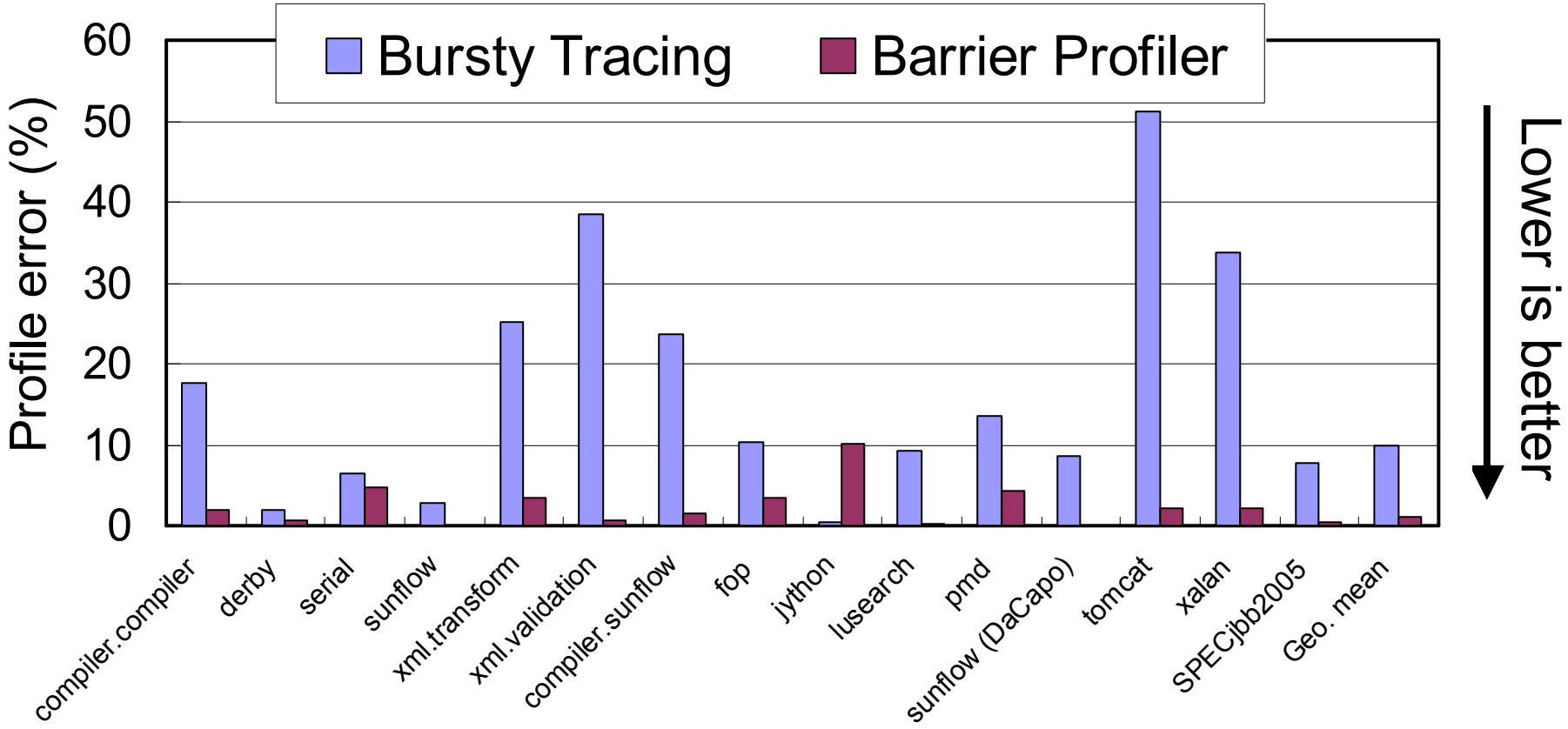
- *Adaptive object sampling:*
  Reduce sampling frequency at the sites that have allocated many uninteresting objects.
- *Adaptive unbarrierization:*
  Stop profiling of objects that no longer satisfy interesting properties.
  - GC-time unbarrierization.
  - Execution-time temporary unbarrierization



© 2012 IBM Corporation

Accuracy (Errors against Perfect Profiles)

- Barrier Profiler mostly resulted in smaller errors than Bursty Tracing.



Errors in estimating the allocation percentage of write-only bytes

© 2012 IBM Corporation