

# Reducing Biased Lock Revocation By Learning

Ian Rogers - Google  
(work done at Azul Systems)  
Balaji Iyengar - Azul Systems

# In Java synchronization is common, so make it fast

- Early approach:
  - Associate a Monitor with an object
  - Locate the Monitor in a table or via a dedicated word in the object header (null  $\Rightarrow$  lack of monitor)

# Thin locking - Bacon bits

- Use word in object header
- Update word using atomic compare-and-swap (CAS) operation
- Object may be:
  - *unlocked*
  - *thin-locked*: owner's Thread ID and a recursive lock count reside in word
  - *fat-locked*: Monitor resides in word
- Extensions:
  - *hashed*: recycle word to hold hash code. Use fat lock for hashed and locked objects
  - *deflation*: reclaim memory for monitors by moving monitor data into object header (done at safepoints to avoid data races)

# Biased locking

- Also known as:
  - Quickly Reacquirable Locks
  - Lock Reservation
- Observations:
  - most objects have a single owner
  - atomic compare-and-swap operations are expensive (memory fence on modern hardware - ie need to flush memory queues)

# Biased locking 2

- On first lock give ownership of lock to a single thread
- Make unlock operation a no-op (if object header is in biased state)
- What about contention?
  - May be false contention as owning thread isn't in a region to hold a lock
  - For true contention the contending thread must wait for the lock anyway

# Biased locking 3

At safepoint owning thread can scan its stack and determine how many times it holds the lock

- compiler stores debug information saying where locks are within a frame
- interpreter has a list of locked objects
- don't allow unbalanced (non-lexically scoped) locks in compiled code

# Revocation is expensive

Revocation must suspend a thread to scan its stack - or ask the thread to do it itself

Common cases such as producer-consumer queues must revoke on change of ownership

Problem not fully appreciated:

- *Kawachiya, Koseki, and Onodera* - Java locks can mostly do without atomic operations - KKO locks - OOPSLA '02
- *Onodera, Kawachiya, and Koseki* - Lock reservation for Java reconsidered - OOPSLA '04

# Let's invent some heuristics

- Don't bias locks for the first 5 seconds when the VM starts
- Bulk rebiasing (Russell and Detlefs, OOPSLA '06):
  - add a bias or don't bias flag to a type
  - on lock entry check whether the type should be biased or not
  - if it should, then bias otherwise:
    - if the lock is unlocked then thin lock
    - otherwise revert to a thin lock (lock entry becomes a safepoint) and lock again
- If GC maintains age information in object header then can have a separate bias flag per age

# Don't believe your benchmarks!

Russell and Detlefs learn what locks have ownership changes during "warm-up" phase of benchmark, then don't speculate during the main benchmark run.

- ie expense of revocation is felt during "warm-up" and not included in the main benchmark run

Other heuristics feel ad hoc and not clear to work in real world.

# Speculative Multiaddress Atomicity

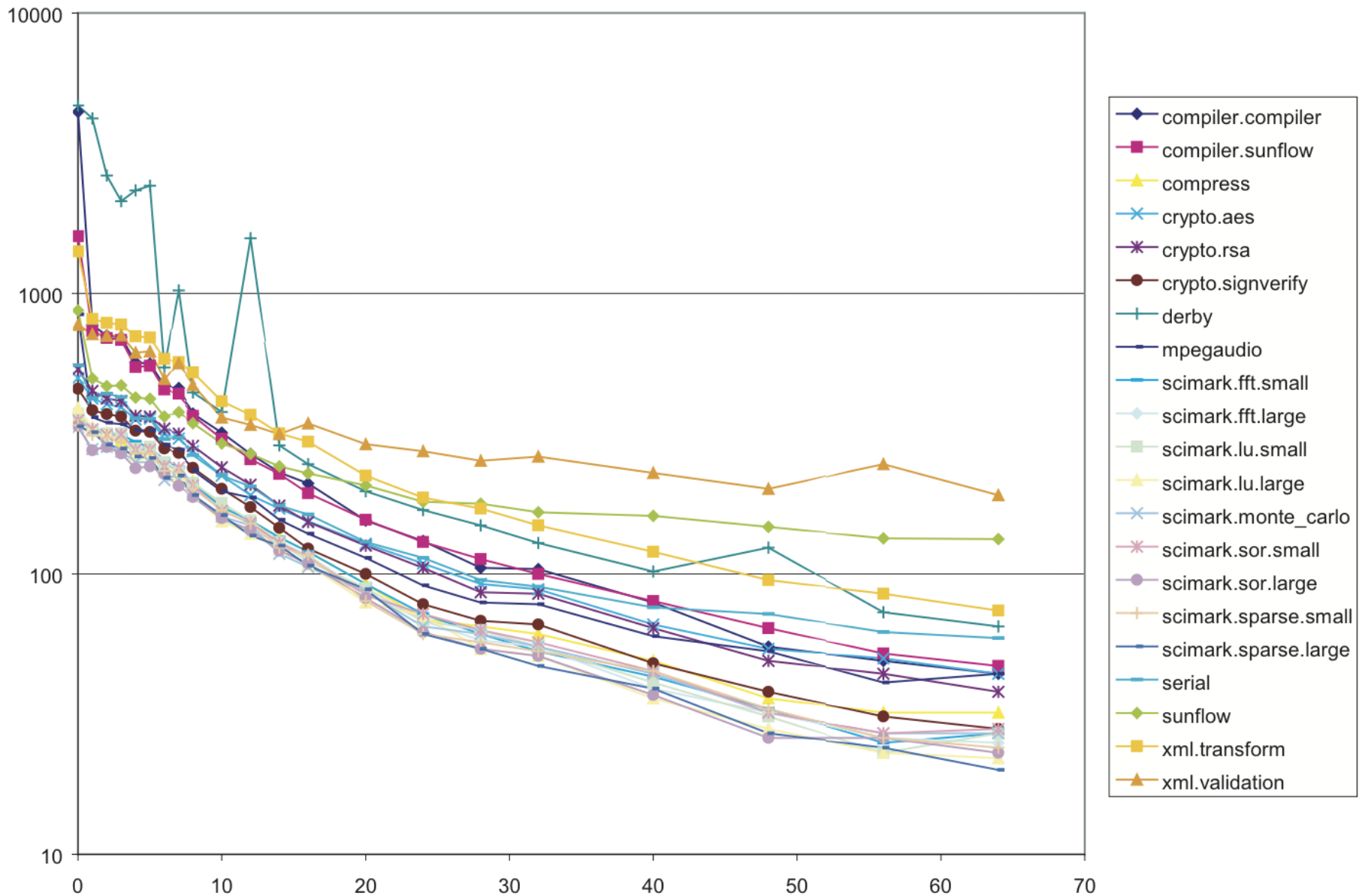
- Azul Systems Hardware Transactional Memory approach
- Fidelity of speculation is per locked object
- Default heuristic is 5 contentions for an object than revert to thin locking

# Rogers-Lock

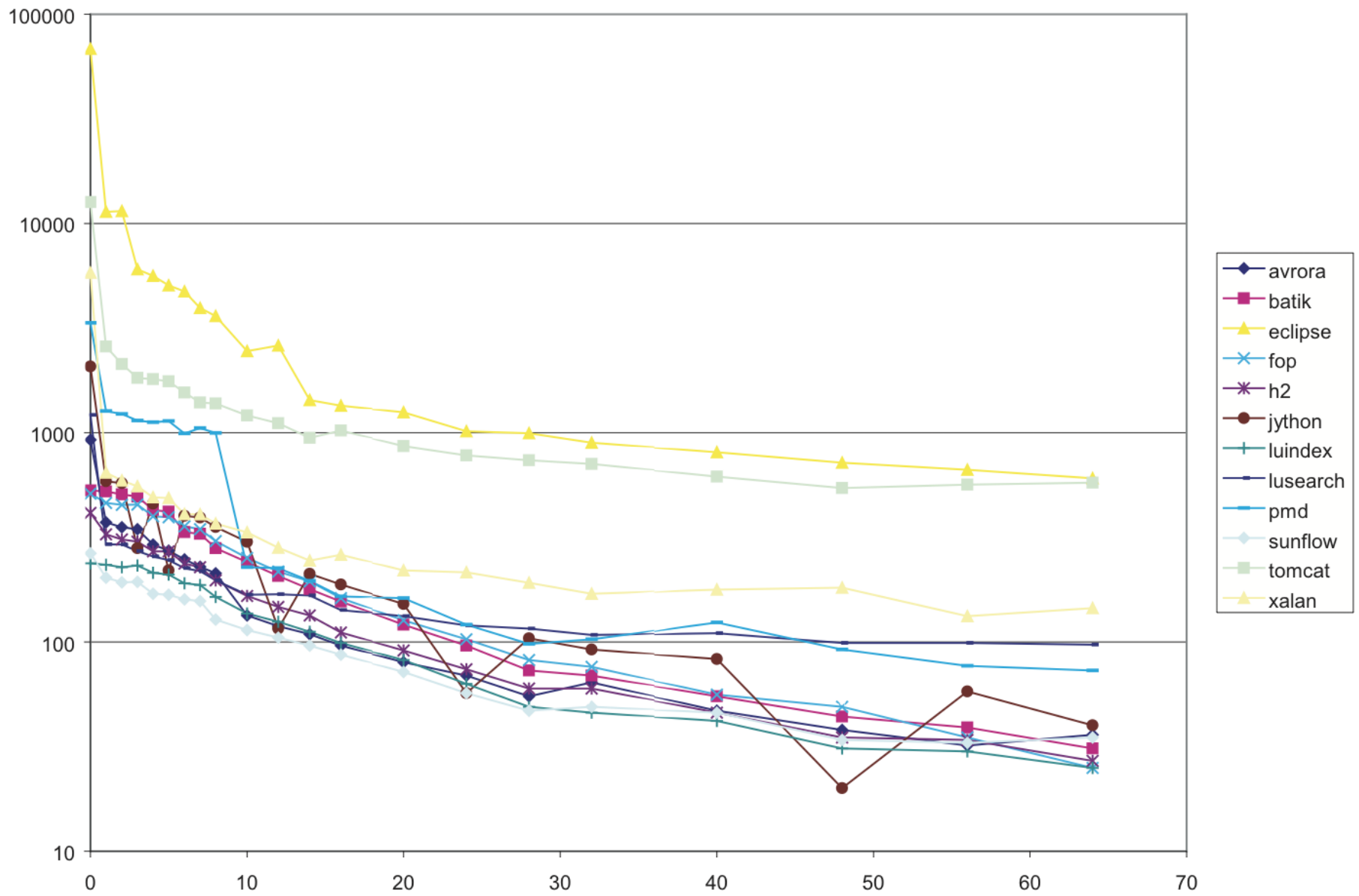
- Use unlocked state to record last owner
  - First lock attempts use thin locking - unlock leaves last owner
  - Change of ownership detected at lock time, set bit in object header to make sure object is never biased
  - After x-many thin lock attempts make lock biased
  - Recursive lock overflow makes lock biased
- 
- Full, production quality state-transition-diagram in paper
    - (no short change)

# Sample window size vs revocations

## SPECjvm2008



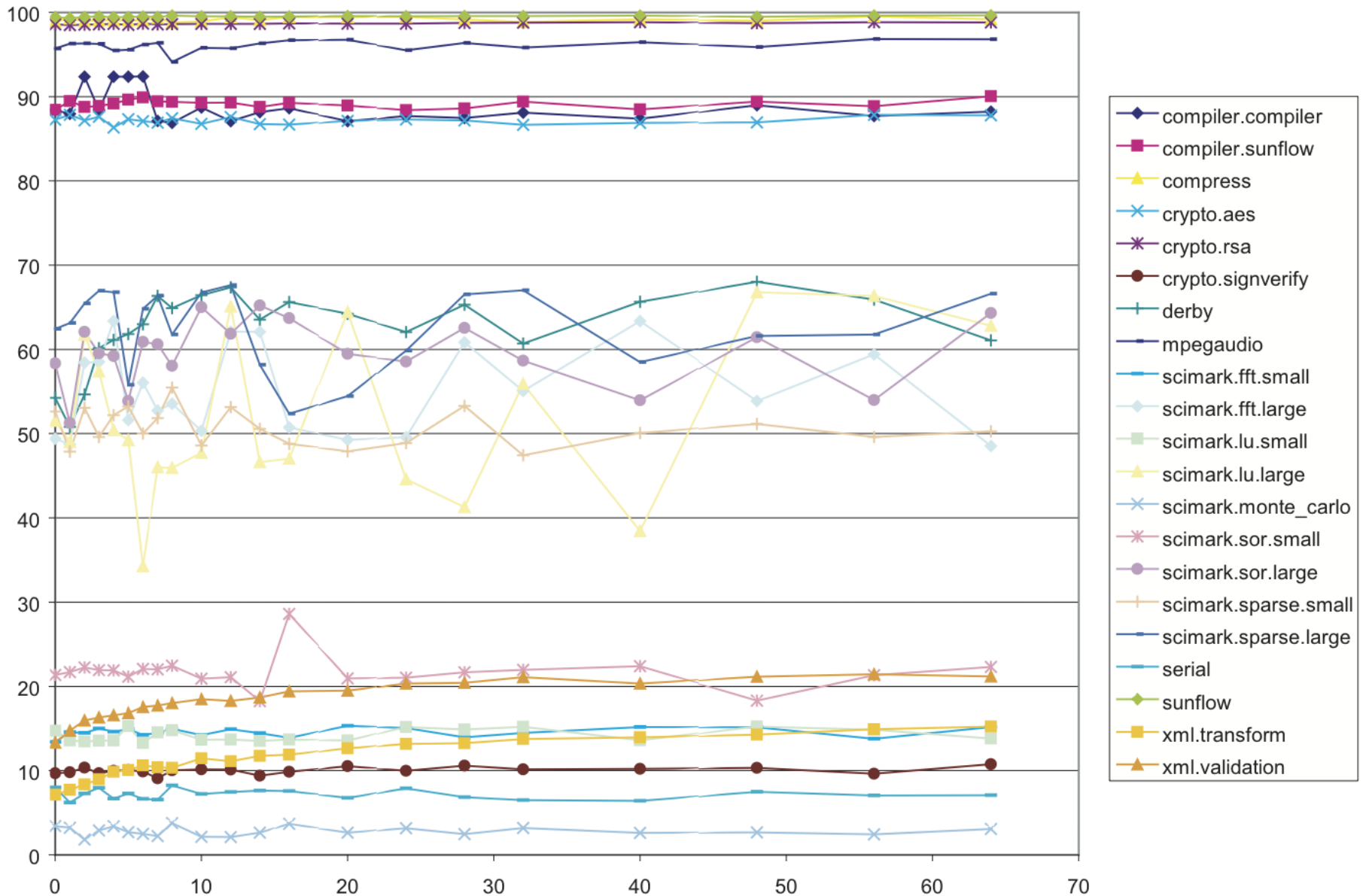
# Sample window size vs revocations DaCapo Bach



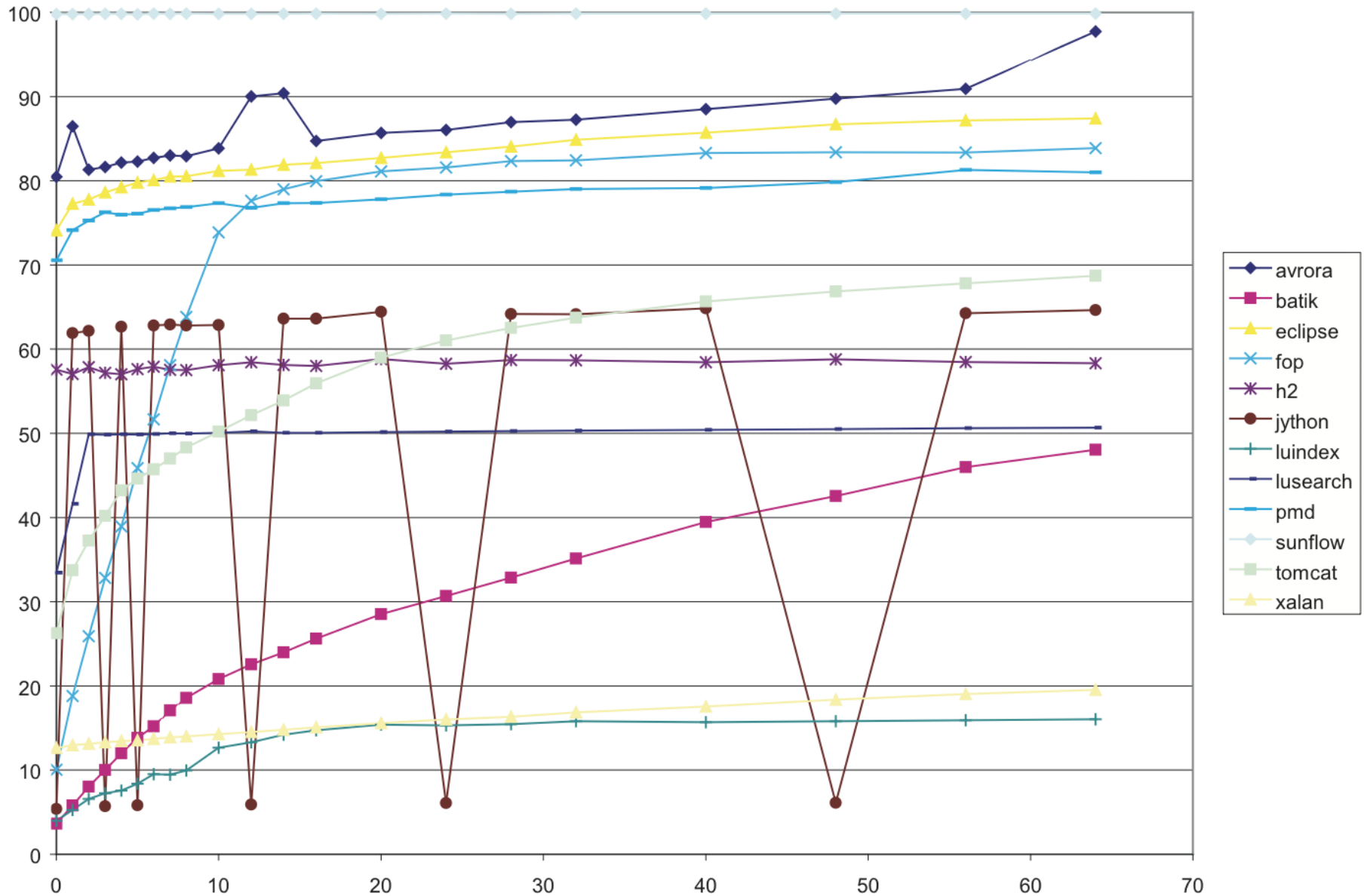
# Effect of sample window size on revocations

More samples mean fewer revocations!

# Sample window size vs percentage of CAS operations - SPECjvm2008



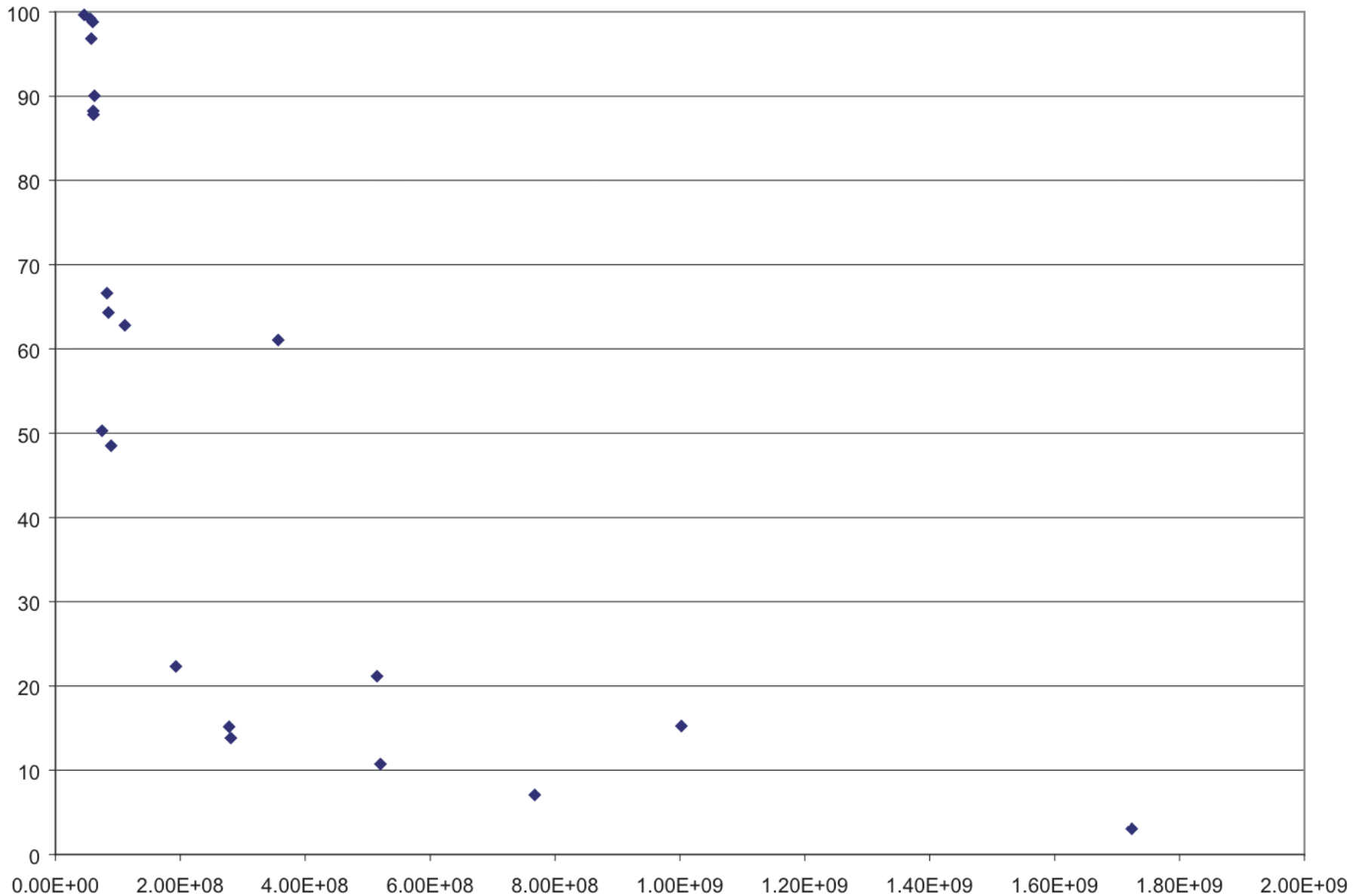
# Sample window size vs percentage of CAS operations - DaCapo Bach



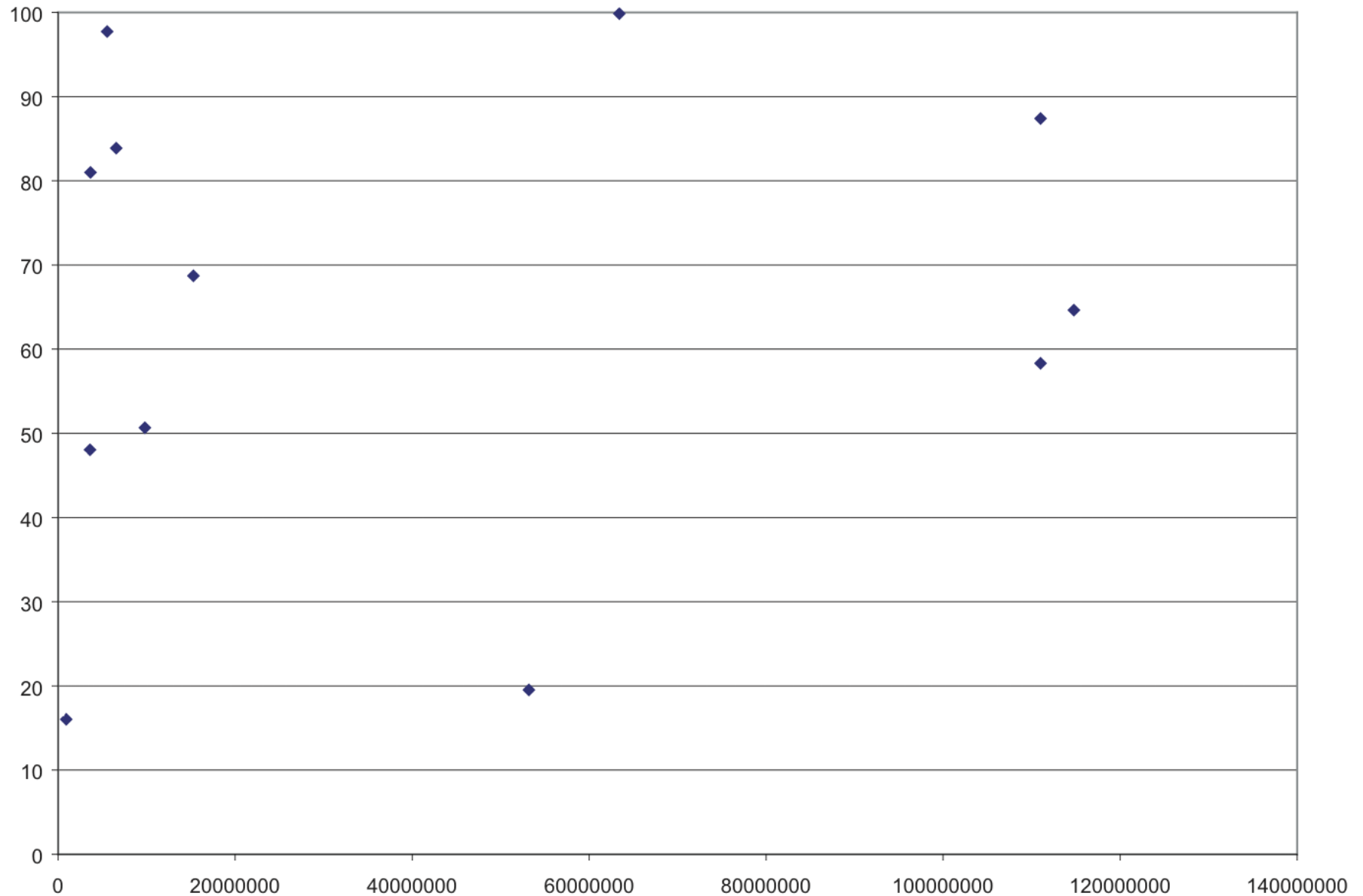
# Effect of sample window size on percentage of CAS operations

Sampling implies more CAS operations, but greater window size doesn't linearly increase the number of CAS operations - the graphs tail off

# Percentage of CAS vs number of lock attempts - SPECjvm2008



# Percentage of CAS vs number of lock attempts - DaCapo Bach



# Percentage of CAS operations vs number of locks attempts

SPECjvm2008 - high percentage of CAS operations implies small number of lock attempts

DaCapo Bach - no clear trend but number of lock attempts in a benchmark run is 15x less than SPECjvm2008

Conclusion: sampling implies an increase in CAS operations compared to directly biasing (such as in KKO locks), but the number of CAS acquisitions is only high as a percentage when the number of lock attempts is small.

# Rogers-Lock improvements

Why learn/sample for types of objects that have never been revoked?

- decrease sample window size as more locks become biased
- increase it as more locks become revoked
- rate of increase/decrease should be in proportion to the cost of a revocation/cost of a biased lock
- ie revocations should increase the sample window size far more than a biased lock attempt reduces it

Further heuristics:

- object allocation site
- lock site
- age of object
- CPU load
- spin lock acquisition attempts

# Conclusions

- Mainstream Java lock implementation (biased locks with bulk revocation) have clear performance pathology which isn't demonstrated by benchmarks due to warm up effect
- Object precision is desirable for fidelity
- Rogers-Lock gives several orders of magnitude improvement over KKO's o2 lock
- CAS operations can be further reduced
- As CAS operations become cheaper relative to the cost of revocation the importance of this approach increases
  - do we already live in that world?
  - Intel's Nehalem architecture made CAS significantly cheaper
- There's a need for an approach that aggregates information and also to investigate other heuristics that play a part in the learning