Performance Considerations in Concurrent Garbage-Collected Systems

Peter Holditch, Chief Architect EMEA, Azul Systems

Presented to JAOO

2008 Garbage Collection Series

About the speaker

Peter Holditch (Chief Architect, EMEA), Azul Systems

- Working with distributed TP systems for nearly 20 years
- Working with java TP systems since WLS 4.0 (9 years ago…)
- Dealing with java application performance / scale problems daily
- Concurrent GC is a must have for this…
  - Can’t scale without it

About Azul

- Azul makes scalable Java Compute Appliances
  - Power Java Virtual Machines on Solaris OS, Linux, AIX, HPUX
  - Scale individual instances to 100s of cores and 100s of GB
  - Production installations ranging from 1GB to 320GB of heap
- All our customers run business-critical java systems aided by our hardware

What's a concurrent garbage collector?

A Concurrent Collector performs garbage collection work concurrently with the application's own execution

A Parallel Collector uses multiple CPUs to perform garbage collection

Agenda

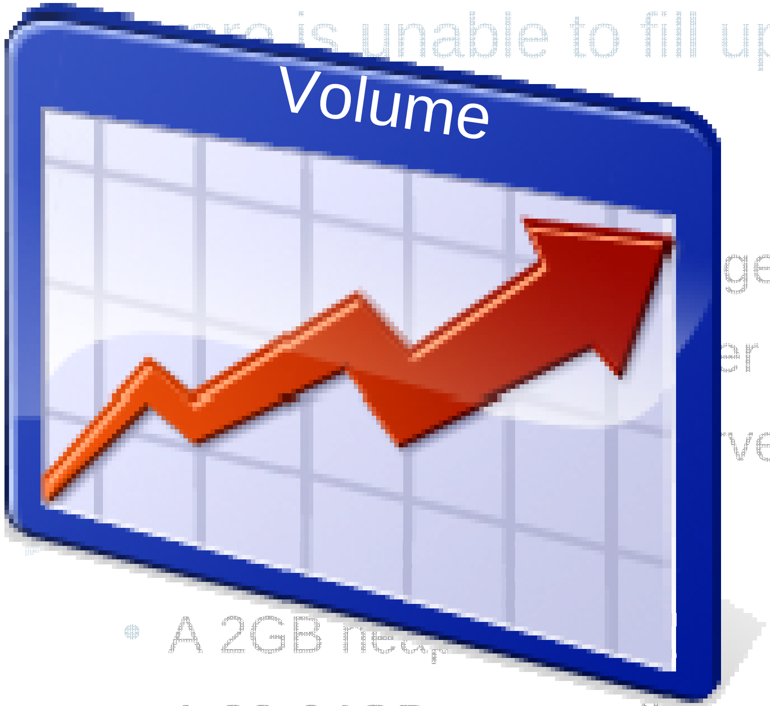
- Background – The big picture
- A load on garbage – The gory details
  - Failure & Sensitivity
  - Terminology & Metrics
  - Detail and inter-relations of key metrics
  - Collector mechanism examples
- Testing Recommendations
- Q & A

Why we really need concurrent collectors

Software is unable to fill up hardware effectively

- 2000:
  - A 512MB heap was “large”
  - A 1GB commodity server was “large”
  - A 2 core commodity server was “large”
- 2008:
  - A 2GB heap is “large”
  - A 32-64GB commodity server is “medium”
  - An 8-16 core commodity server is “medium”
- The erosion started in the late 1990s

Why we **really** need concurrent collectors



- are is unable to fill up hardware effectively
- ge”
- er was “large”
- rver was “large
  - A 2GB hea
  - A 32-64GB commodity server is “me
  - An 8-16 core commodity server is “n
- The erosion started in the late 19

Benefits for trading platforms

CALYPSO

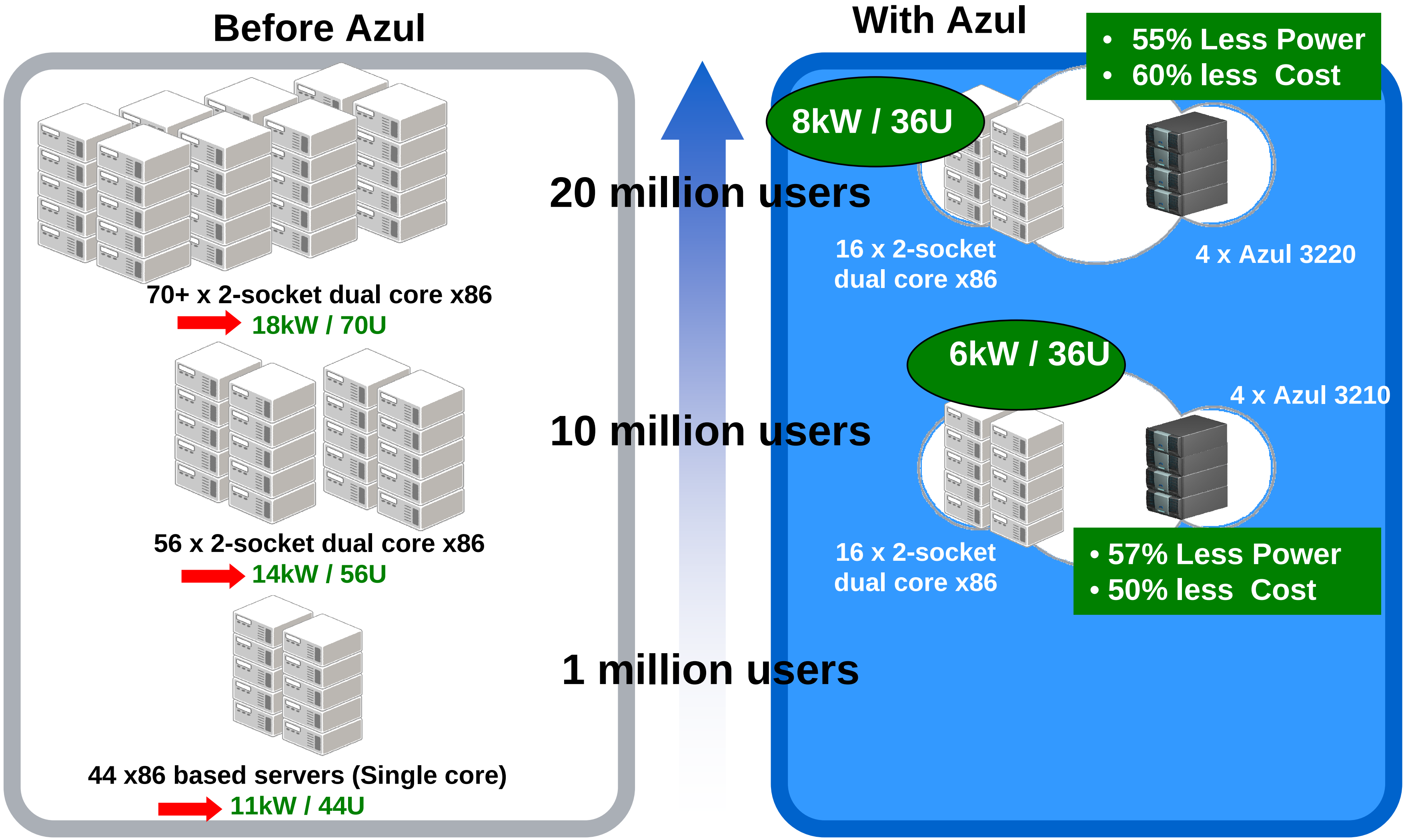
User	Issue	Azul benefit to Data Server	Benefit
NY Investment Bank #1	Trading volumes peak at 156k conc > 10 se tim	Heap size increased	Trading volume increased .6M concurrent response times
NY Investment Bank #2	Ba tra ho Sta GC live		ion reduced s eporting data ng throughput bility and consistency
UK Investment Bank #1	4-h job Lin co		ed by 4X to eporting data ng throughput
UK Investment bank #2	End-of to 150k tr Trading vo trades / second 3 minute peak GC pauses with 10 GB heap	GC pauses reduced from 3 mins to < 1 second	clearing volume d 2X to 300k trades volume increased 2X to 12 trades / second Fast, consistent response times

10x increase in trading volume
3-4x shorter batch duration
2x greater clearing volume

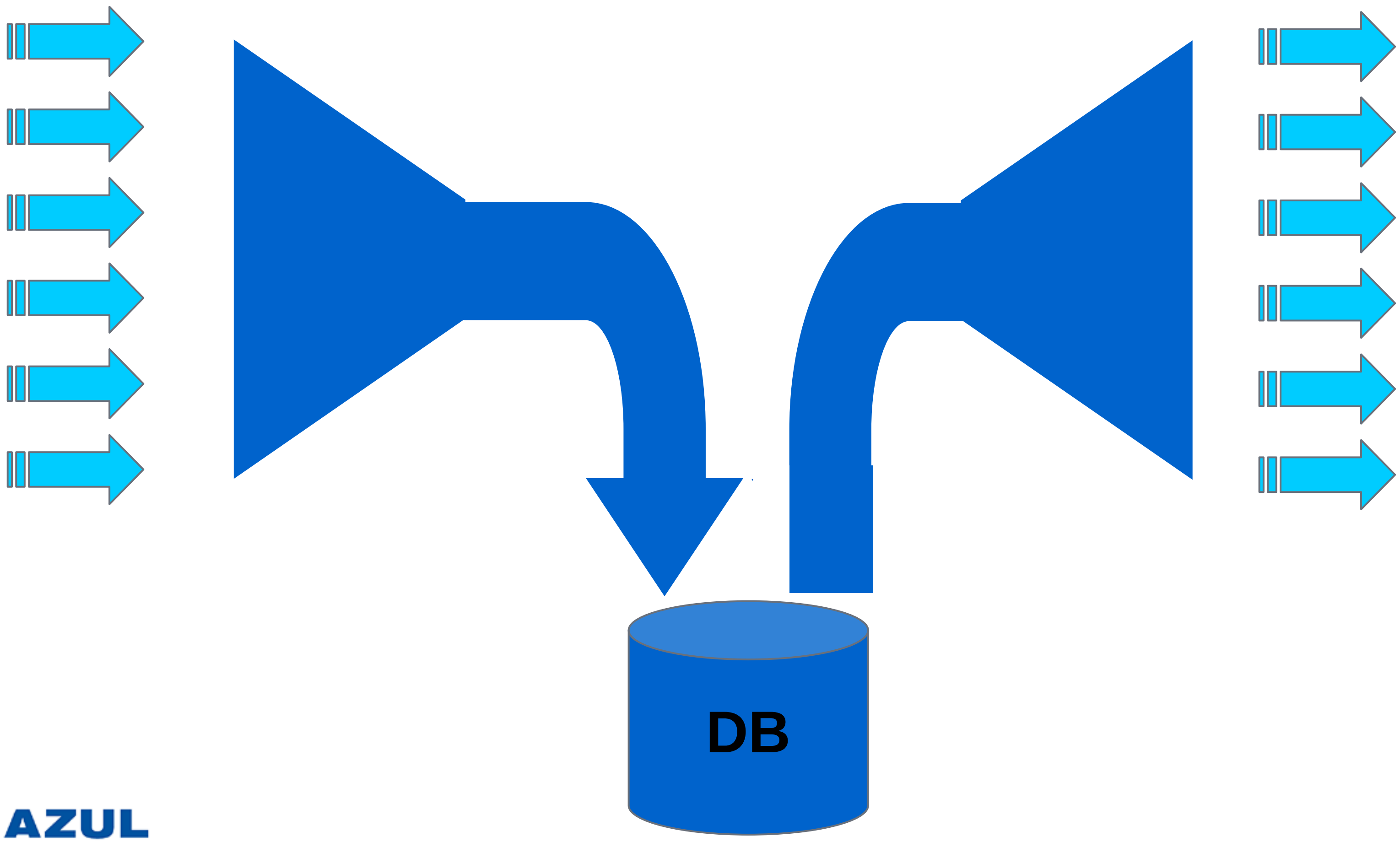
Ability to run on-line processing and end of day concurrently

Azul uniquely delivers these benefits
with no application changes
(and in a reduced datacentre footprint)

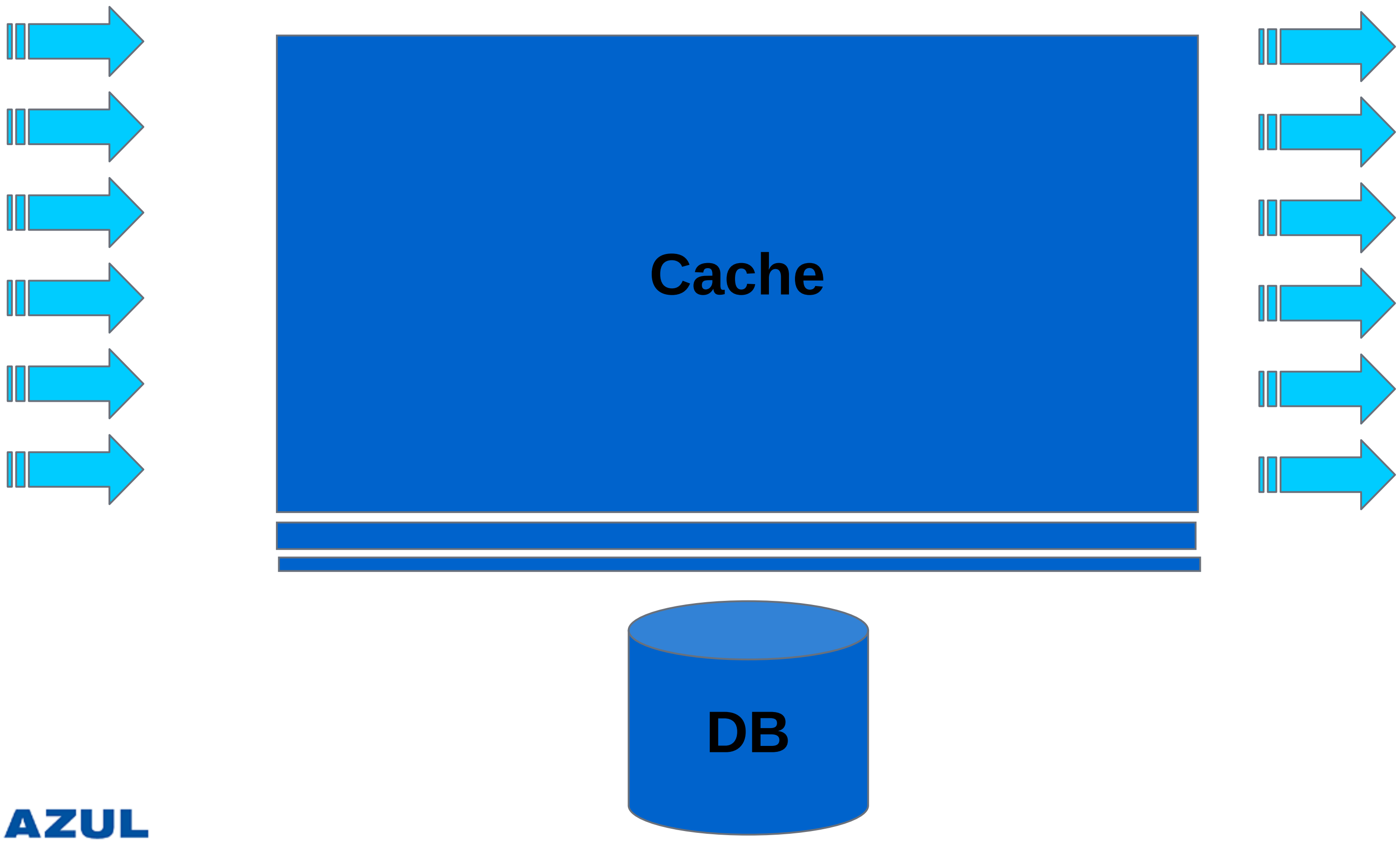
Scale Without Sprawl



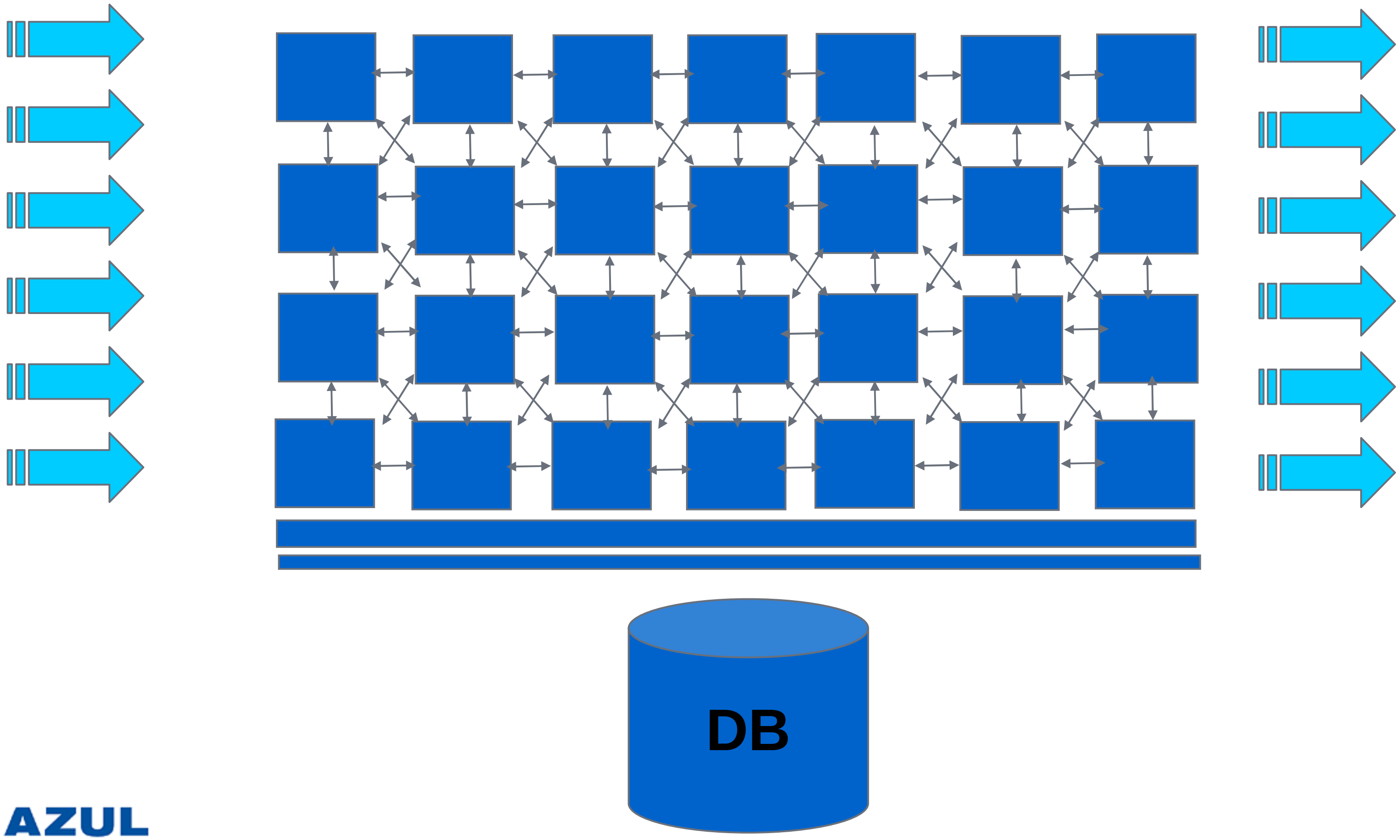
High throughput, large dataset problems



High throughput, large dataset problems



High throughput, large dataset problems



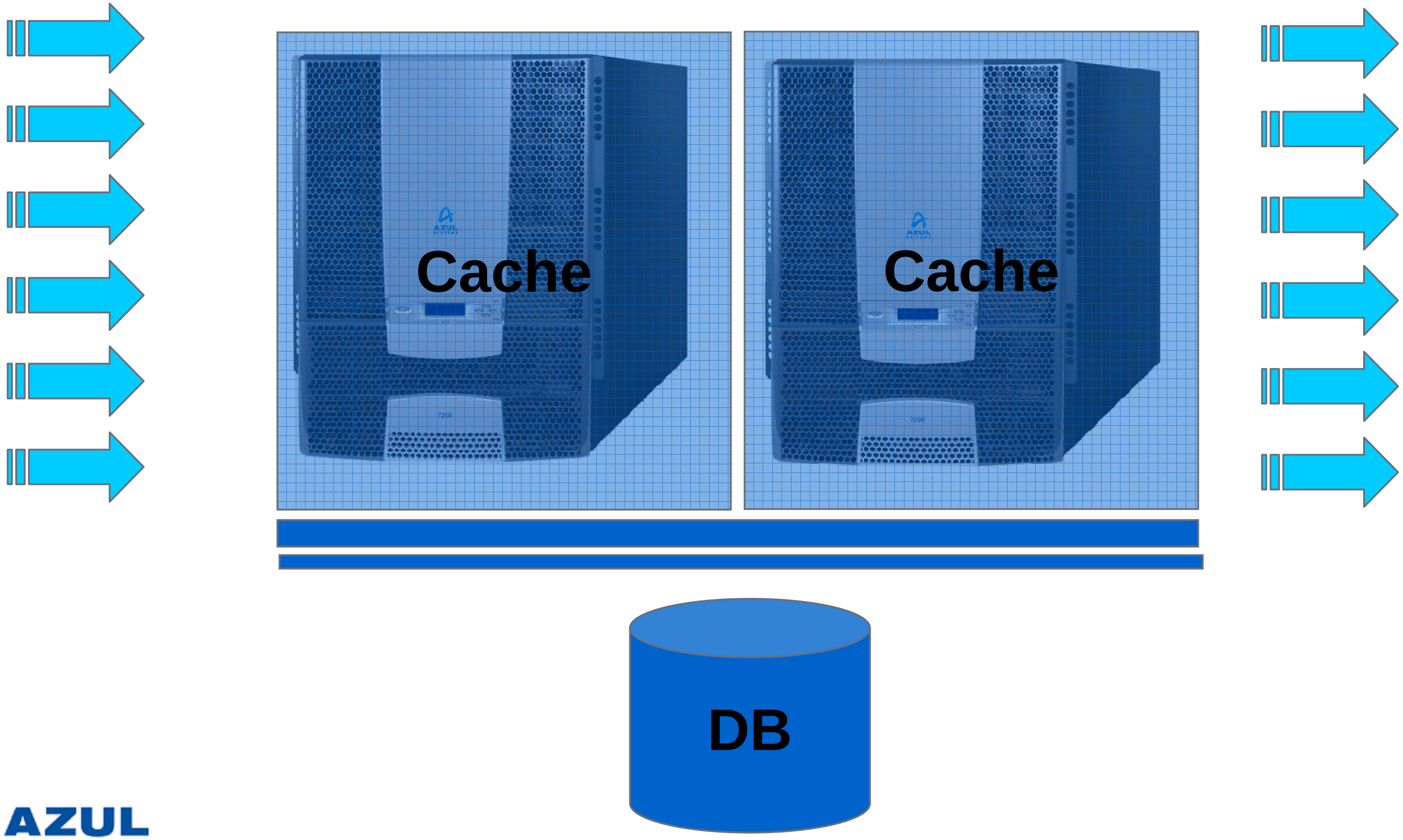
High throughput, large dataset problems

GigaSpaces

GEMFIRE THE ENTERPRISE DATA FABRIC

TERRACOTTA

ORACLE Coherence



Agenda

- Background – The big picture
- A load on garbage – The gory details
  - Failure & Sensitivity
  - Terminology & Metrics
  - Detail and inter-relations of key metrics
  - Collector mechanism examples
- Testing Recommendations
- Q & A

What constitutes "failure" for a collector?

It's not just about correctness any more

- A Stop-The-World collector fails if it gets it wrong…
- A concurrent collector [also] fails if it stops the application for longer than requirements permit
  - "Occasional pauses" longer than SLA allows are real failures
    - Even if the Application Instance or JVM didn't crash
  - Otherwise, you would have used a STW collector to begin with
- Simple example: Clustering
    - Node failover must occur in X seconds or less
    - A GC pause longer than X will trigger failover. It's a fault.
      (If you don't think so, ask the guy whose pager just went off…)

Concurrent collectors can be sensitive

Go out of the smooth operating range, and you'll pause

- Correctness now includes response time
- Just because it didn't pause under load X, doesn't mean it won't pause under load Y
- Outside of the smooth operating range:
  - More state (with no additional load) can cause a pause
  - More load (with no additional state) can cause a pause
  - Different use patterns can cause a pause
- Understand/Characterize your smooth operating range

Terminology

Useful terms for discussing concurrent collection

- Mutator
  - Your program…
- Parallel
  - Can use multiple CPUs
- Concurrent
  - Runs concurrently with program
- Pause time
  - Time during which mutator is not running any code
- Generational
  - Collects young objects and long lived objects separately.
- Promotion
  - Allocation into old generation
- Marking
  - Finding all live objects
- Sweeping
  - Locating the dead objects
- Compaction
  - Defragments heap
  - Moves objects in memory
  - Remaps all affected references
  - Frees contiguous memory regions

Metrics

Useful metrics for discussing concurrent collection

- Heap population (aka Live set)
  - How much of your heap is alive
- Allocation rate
  - How fast you allocate
- Mutation rate
  - How fast your program updates references in memory
- Heap Shape
  - The shape of the live object graph
  - * Hard to quantify as a metric...
- Object Lifetime
  - How long objects live
- Cycle time
  - How long it takes the collector to free up memory
- Marking time
  - How long it takes the collector to find all live objects
- Sweep time
  - How long it takes to locate dead objects
  - * Relevant for Mark-Sweep
- Compaction time
  - How long it takes to free up memory by relocating objects
  - * Relevant for Mark-Compact

Cycle Time

How long until we can have some more free memory?

- Heap Population (Live Set) matters
  - The more objects there are to paint, the longer it takes
- Heap Shape matters
  - Affects how well a parallel marker will do
  - One long linked list is the worst case of most markers
- How many passes matters
  - A multi-pass marker revisits references modified in each pass
  - Marking time can therefore vary significantly with load

Heap Population (Live Set)

It’s not as simple as you might think…

- In a Stop-The-World situation, this is simple
  - Start with the “roots” and paint the world
  - Only things you have actual references to are alive
- When mutator runs concurrently with GC:
  - Not a “snapshot” of a single program state
  - Objects allocated during GC cycle are considered “live”
  - Objects that die after GC starts may be considered “live”
  - Weak references “strengthened” during GC…
- So assume:
  - Live_Set >= STW_live_set + (Allocation_Rate * Cycle_time)

Mutation rate

Does your program do any real work?

- Mutation rate is generally linear to work performed
  - The higher the load, the higher the mutation rate
- A multi-pass marker can be sensitive to mutation:
  - Revisits references modified in each pass
  - Higher mutation rate → longer cycle times
  - Can reach a point where marker cannot keep up with mutator
  - e.g. one marking thread vs.15 mutator threads
- Some common use patterns have high mutation rates
  - e.g. LRU cache

Object lifetime

Objects are active in the Old Generation

- Most **allocated** objects do die young
  - So generational collection is an effective filter
- However, most **live** objects are old
  - You’re not just making all those objects up every cycle…
- Large heaps tend to see real churn & real mutation
  - e.g. caching is a very common use pattern for large memory
- OldGen is under constant pressure in the real world
  - Unlike some/most benchmarks (e.g. SPECjbb)

Major things that happen in a pause

The non-concurrent parts of “mostly concurrent”

- If collector does Reference processing in a pause
  - Weak, Soft, Final ref traversal
  - Pause length depends on # of refs.
  - Sensitive to common use cases of weak refs
    - e.g. LRU & multi-index cache patterns
- If the collector marks mutated refs in a pause
  - Pause length depends on mutation rate
  - Sensitive to load
- If the collector performs compaction in a pause…

More things that may happen in a pause

More “mostly concurrent” secrets

- When collector does Code & Class things in a pause
  - Class unloading, Code cache cleaning, System Dictionary, etc.
  - Can depend on class and code churn rates
  - Becomes a real problem if full collection is required (PermGen)
- GC/Mutator Synchronization, Safe Points
  - Can depend on time-to-safepoint affecting runtime artifacts:
    - Long running no-safepoint loops (some optimizers do this).
    - Huge object cloning, allocation (some runtimes won’t break it up).
- Stack scanning (look for refs in mutator stacks)
  - Can depend on # of threads and stack depths

Fragmentation & Compaction

You can't delay it forever

- Fragmentation *will* happen
  - Compaction can be delayed, but not avoided
  - "*Compaction is done with the application paused. However, it is a necessary evil, because without it, the heap will be useless…*" (JRockit RT tuning guide).
- If Compaction is done as a stop-the-world pause
  - It will generally be your worst case pause
  - It is a likely failure of concurrent collection
- Measurements without compaction are meaningless
  - Unless you can prove that compaction won't happen (Good luck with that)

Example: HotSpot CMS

Collector mechanism examples

- Stop-the-world compacting new gen (ParNew)
- Mostly Concurrent, non-compacting old gen (CMS)
  - Mostly Concurrent marking
    - Mark concurrently while mutator is running
    - Track mutations in card marks
    - Revisit mutated cards (repeat as needed)
    - Stop-the-world to catch up on mutations, ref processing, etc.
  - Concurrent Sweeping
  - Does not Compact (maintains free list, does not move objects)
- Fallback to Full Collection (Stop the world, serial).
    - Used for Compaction, etc.

Example: Azul GPGC

Collector mechanism examples

- Concurrent, compacting new generation
- Concurrent, compacting old generation
- Concurrent guaranteed-single-pass marker
  - Oblivious to mutation rate
  - Concurrent ref (weak, soft, final) processing
- Concurrent Compactor
  - Objects moved without stopping mutator
  - Can relocate entire generation (New, Old) in every GC cycle
- No Stop-the-world fallback
  - Always compacts, and does so concurrently

Agenda

- Background – The big picture
- A load on garbage – The gory details
  - Failure & Sensitivity
  - Terminology & Metrics
  - Detail and inter-relations of key metrics
  - Collector mechanism examples
- Testing Recommendations
- Q & A

Measurement Recommendations

When you are actually interested in the results…

- Measure application – not synthetic tests
  - Garbage in, Garbage out
- Avoid the urge to tune GC out of the testing window
  - You're only fooling yourself
  - Your application needs to run for more than 20 minutes, right?
  - Most industry benchmarks are tuned to avoid GC during test ☹
- Rule of Thumb:
  - You should see 5+ of the "bad" GCs during test period
  - Otherwise, you simply did not test real behavior
  - Test until you can show it's stable (e.g. What if it trends up?)
  - Believe your application, not -verbosegc

Measurement Techniques

Make reality happen

- Aim for 20-30 minute “stable load” tests
  - If test is longer, you won’t do it enough times to get good data
  - Don’t “ramp” load during test period – it will defeat the purpose
  - We want to see several days worth of GC in 20-30 minutes
- Add low-load noise to trigger “real” GC behavior
  - Don’t go overboard
  - A moderately churning large LRU cache can often do the trick
  - A gentle heap fragmentation inducer is a sure bet
  - Can easily be added orthogonally to application activity
  - See Azul’s “Fragger” example (http://e2e.azulsystems.com)

Establish smooth operating range

Know where it works, and know where it doesn't…

- Test main metrics for sensitivity
- Stress Heap population, allocation, mutation, etc.
- Add artificial load-linear stress if needed
  - E.g. Increase allocation and mutation per transaction
  - E.g. Increase state per session, increase static state
  - E.g. Increase session length in time
  - Drive load with artificially enhanced GC stress
  - Keep increasing until you find out where GC breaks SLA in test
  - Then back off and test for stability

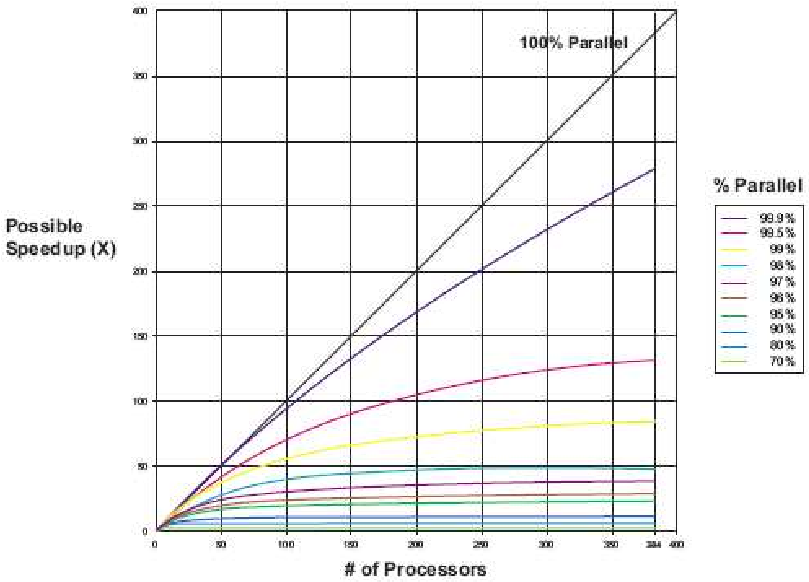
Summary

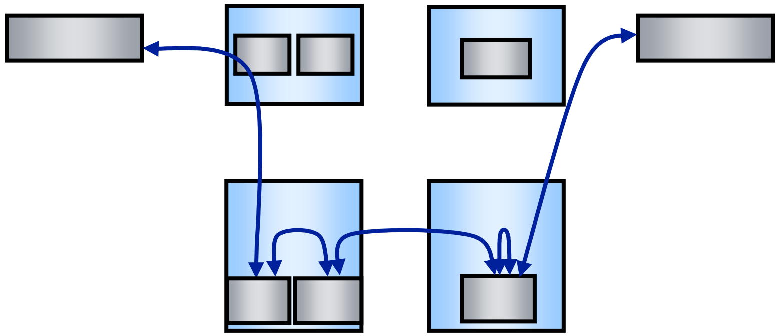
Know where the cliff is, then stay away from the edge…

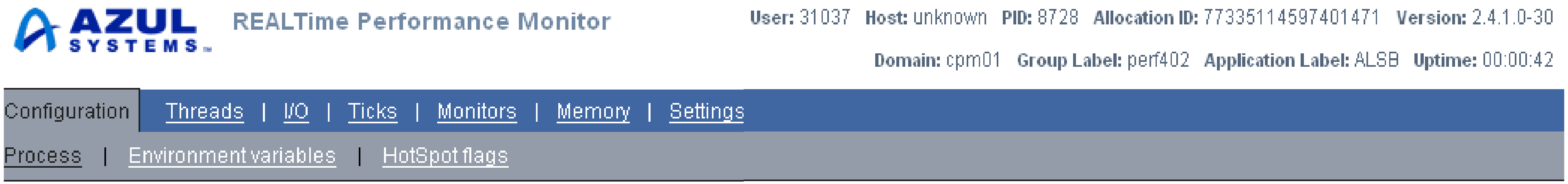
- Sensitivity is key
  - If it fails, it will be without warning
- Know where you stand on key measurable metrics
  - Application driven: Live Set, Allocation rate, Heap size
  - GC driven: Cycle times, Compaction Time, Pause times
- Deal with robustness first, and only then with efficiency
  - But efficient and 2% away from failure is not a good thing
- Establish your envelope
  - Only then will you know how safe (or unsafe) you are

http://e2e.azulsystems.com

Other Azul application scale enablers …







Q&A

Performance Considerations in Concurrent Garbage–Collected Environments

Peter Holditch, Chief Architect EMEA,
Azul Systems

www.azulsystems.com
peter.holditch@azulsystems.com

Tak!

If you have further questions… Please visit our booth (en route to kammermusik sal, rytmisk sal)

Peter Holditch, Chief Architect EMEA,
Azul Systems

www.azulsystems.com
peter.holditch@azulsystems.com