UKSUG
SAP DATABASE & TECHNOLOGY USER GROUP

Sponsored by

aws

TechSelect 2018
SAP Database & Technology User Show

Sponsored by

SAP

ASE 16.0 HugePages & Memscale
A Case Study

Joe Woodhouse, Prima Donna Consulting

June 26, 2018

aws marketplace

THE BEST RUN SAP

Summary

About the author

Part A – RHEL Huge Pages

Part B – ASE Memscale

Appendix A – Huge Pages – Details

Appendix B – Memscale – Details

About the author

Sybase Australia 1996 – 2003.

Freelance consultant via Prima Donna Consulting for 15+ years.

Works exclusively with SAP ASE, SAP IQ, and SAP Replication Server, for 23+ years.

Based in London, UK and Melbourne, Australia.

International Sybase User Group Board of Directors since 2010.

Not a lawyer – no charge for emails!

- joe.woodhouse@primadonnaconsulting.com

Part A – RHEL Huge Pages

Introduction – RHEL Huge Pages

O/S memory page size is configurable in Red Hat Enterprise Linux.

- Default is 4Kb (= Huge Pages ***disabled***).
- Larger size is available on most modern Intel x64 CPUs.
- SAP recommends 2Mb Huge Pages for ASE. (And so do most other database vendors.)
  - Used to be necessary for >256Gb RAM, but not in any recent version of RHEL.
  - SAP claims 8-10% performance improvement by using it.

Seems like a no-brainer – who doesn't want +10% performance by tuning one thing?

Note: this is *not* the same as RHEL Transparent Huge Pages.

- SAP and other vendors agree this should be ***disabled*** for database servers.
- Alas, it is enabled by default.

Huge Pages – why not?

The correct Huge Pages setting depends on physical RAM and on total ASE “max memory”.

- Trivial when you only have a few servers.
- Problematic when you have thousands.

Many Linux configuration management tools enforce “one size fits all”.

- This makes RHEL Huge Pages difficult or impossible in large enterprises.

A very large client asked for help:

- Their benchmarks found Huge Pages was at best a 3% difference.
- 3% wasn’t worth the administrative headaches of RHEL configuration management.
- SAP claims of 8-10% were troubling.

We designed a synthetic benchmark to settle this once and for all.

Huge Pages – findings & recommendations

Surprise #1: Transparent Huge Pages helped a bit (~3.9%).

Surprise #2: “Regular” Huge Pages (2Mb page size) helped less (~1%).

No surprise: “Extra Huge Pages” (1Gb page size!) helped the most (~5.6%).

Client said they could sacrifice 5.6% performance to keep configuration management simple.

- Excellent example of why the “technically obvious solution” isn’t always the best answer.

Time is short; many more details and references in Appendix A.

Part B – ASE Memscale

Introduction – ASE Memscale

SAP ASE includes many optional features sold as separate premium licenses.

- ASE 16.0 introduced Memscale as a premium licensed option. (Includes IMDB option.)

Lots of scepticism and cynicism by both technical and business audiences.

- And it’s true: it’s not cheap.

This presentation intends to convince you that if you need it, you can afford it.

- Also true: if you can afford it, you probably need it.

To do that, we must show you:

- When you need it.
- What it will do for you.

These are strong claims, and strong claims need strong evidence.

- We report on a real case study, at a real user site, with the numbers.
- We break down which Memscale features did and did not help there, and why.

Agenda

Two problems

Inside the database

ASE 16.0 Memscale

How we benchmarked

Findings

Recommendations

For further investigation

Appendix B – Detailed Stats and Analysis

Two problems

Let’s talk about ***scalability***:

IT scalability is a measure of ROI on IT infrastructure.

“We bought bigger/faster hardware…”

- “… did it fix the problem?”
- “… did it do so cost-effectively?”
  - “… we doubled the hardware – did we double the performance?”

Any ceiling or limit eventually kills scalability.

- After a point spending more money does nothing – or may even make it worse.

Let’s talk about ***contention***:

Contention is a serious problem for a database.

- Faster CPUs / memory / disks / network don’t help.
- More memory doesn’t help.
- More CPUs ***always*** makes it worse.
- Rewriting SQL only sometimes helps.
- Data volumes / transaction rates / user counts are usually increasing.

This means that contention will eventually kill any hope of scalability.

Inside the database (1)

Contention in a database means some form of locking on a ***physical*** shared resource.

- Not the same as logical locking needed for database transactional consistency.

Two or more threads changing the same thing at the same time = corruption.

- Physical locks guard shared resources.
- They protect the shared resource from simultaneous changes.

Shared resources usually means something in memory.

- BTW this is why more/faster CPUs won’t help.
- CPUs are already much faster than memory.
- If something slow is blocking something fast, making the fast thing even faster will never help.

Inside the database (2)

Contention on any shared resource can only be eased in four basic ways:

- Divide & conquer – partition or split one lock into many.
- Get faster – fix the slowest part of the internal process.
- Something for everyone – add a private or local resource to each thread.
- Stop using locks – find a fundamentally different way of doing something.

Over the years ASE has reduced contention in most areas.

- Each major release usually included multiple examples of each of the above.
- As of ASE 15.7 the current bottleneck is ***spinlock contention***.

ASE 15.7 can be pushed to some limits that cannot be exceeded by any means.

- ASE 16.0 out of the box (no Memscale) eases or removes a few of those limits.
- ASE 16.0 **SP02** Memscale fixes more.
- ASE 16.0 **SP03** Memscale fixes most of the rest.

Inside the database (3)

Spinlocks are pure-memory short-duration locks.

- Cannot be held from one ASE context switch to the next - so they are very fast.
- Any thread waiting for a spinlock doesn’t wait or sleep.
- It “sits and spins”, does a lot of nothing very fast, which burns CPU.
  - (NOOP is still a CPU operation.)
- This consumes CPU and raises CPU utilisation.

The costs and implications of spinlock contention aren’t necessarily obvious.

Consider two scenarios (next slide).

Inside the database (4)

Scenario 1: spinlock contention causes CPU consumption.

- So IT added more CPUs (since that’s the usual fix for high CPU utilisation).
- But more CPUs make spinlock contention worse.
- … therefore driving up CPU utilisation even further.
- All that CPU is now unavailable for actually useful work.
- So everything else gets slower.

Scenario 2: IT adds more memory.

- The same spinlocks have to protect more memory each.
- So spinlock contention increases.
- Meanwhile fitting more of the database in memory means less physical I/O to disk.
- … so CPU utilisation goes up (“in-memory table scans”).
- So spinlock contention increases even more.

Technology problem = business problem

What all this means is that in both scenarios IT spent more money on hardware.

- But performance either didn’t get better… or it got worse.

What it also means is that once all other factors are addressed:

- Spinlock contention is now the limiting factor…
- And spinlock contention ***cannot*** be fixed by spending more money on hardware.

Two major problems!

- Poor ROI when IT hardware spend does not gain the expected benefits.
- Unable to do anything to improve performance no matter how much is spent.

ASE 16.0 Memscale (1)

ASE 16.0 Memscale features were introduced to address these ROI and performance problems.

Actually some features are included “for free” with ASE 16.0.x:

- Transaction log spinlock contention:
  - “user log cache queue size” (use this instead of “user log cache size”).
- Procedure cache spinlock contention:
  - “engine local cache percent”.
  - “enable large chunk elc” (use this instead of –T753).
  - “large allocation autotune”.
- Metadata cache spinlock contention: internal ASE rewrites, not configurable.
- Latch contention: internal ASE rewrites, not configurable.
- Data cache spinlock contention:
  - “lockless data cache”.

ASE 16.0 Memscale (2)

The remaining ASE 16.0 Memscale features require the separate Memscale premium license.

The other Memscale features can & should be separately enabled & configured:

- Transactional memory – leverage Intel CPU hardware optimisations.
- Native compiled query plans – compile in OS rather than in ASE.
- Latch-free B-tree indexes – if latches are causing contention, get rid of them.
- In-memory row-store – cache database rows, not pages, and auto-tune.
- Hash-based indexing – cache faster indexes for cached rows (requires IMRS).
- Multi-version concurrency control – completely rewrite database locking.
- NVCache – leverage fastest possible PCIe/NVRAM hardware as extension to data cache.

ASE 16.0 Memscale (3)

Useful to group the ASE 16.0 Memscale features by ease of implementation:

Group A – trivial to implement, suitable for every qualifying environment:

- Transactional memory (“TSX”).
  - Linux only! Intel x64 only, and only recent CPUs.

Group B – simple to implement, no changes to database schema or application code:

- Native compiled query plans (“SNAP”).
  - Note many restrictions on when it can be used, which may mean even when enabled it may not be used.
  - Linux only! Requires PCI database.
- Lockless data cache (“LLDC”).

Group C – moderate complexity, customise per environment:

- Configurable parameters for user log cache and procedure cache.
- Latch-free B-tree indexes (“LFB”).

ASE 16.0 Memscale (4)

Useful to group the ASE 16.0 Memscale features by ease of implementation:

Group D – requires major implementation:

- In-memory row-store (“IMRS”).
- Hash-based indexing (“HCB”) – requires IMRS.

Group E – requires application code rewrite:

- Multi-version concurrency control (“MVCC”).

Group F – requires new dedicated hardware:

- NVCache.

How we benchmarked (1)

For this case study, we benchmarked items in groups A – D only.

Our initial efforts used the ASE 16.0 Workload Analyzer (“WA”):

- Can capture all work done in ASE during a sample interval, and then repeatedly replay it.
- Intention was to capture real-world application workload and replay for benchmarks.
- Unfortunately we ran into a WA issue with login authentication that was not ASE-based.

Focus then switched to a “synthetic” benchmark.

- Java-based trading simulation.
- Normally synthetic benchmarks are inferior to real workload.
- This was still appropriate as we wanted only to deliberately provoke spinlock contention.

We would add Memscale features (configured & tuned) one at a time.

How we benchmarked (2)

Initial baseline (800 concurrent users hammering ASE 16.0 SP03):

- ~86% spinlock contention in most utilised data cache.
- ~90% ASE CPU busy (ASE was using 128 CPUs).
- This ASE was in trouble! If this were ASE 15.7 then it would be the end of the road.

All benchmarks began with a “PROD-worthy” ASE configuration.

- Two further configurations evolved during testing.

Two broad sets of benchmark runs:

- Set 1 – 800 concurrent users, realistic wait times between new trades or requests.
  – Began with ASE config A, moved to ASE config B.
- Set 2 – 800 concurrent users, unrealistically reduced (50ms) waits to increase load on ASE.
  – Used new ASE config C guided by results of first set of benchmarks.

How we benchmarked (3)

What we measured:

- Spinlock contention:
  - Chief metrics were data cache spinlock contention as reported by sp_sysmon.
  - We measured default data cache, system tables cache, and log cache.
  - Expected results: spinlock contention will fall as Memscale features are enabled.
- ASE CPU utilisation:
  - As reported by sp_sysmon under “Total Server Busy %”.
  - Expected results: CPU utilisation will fall as spinlock contention falls.
- Throughput:
  - As reported by sp_sysmon “Transactions (xacts) per ASE CPU unit”.
  - Expected results: throughput will rise as spinlock contention falls.

Findings (1)

Broadly speaking all expectations were confirmed:

- Spinlock contention fell significantly with Memscale.
- Spinlock contention fell only with Memscale features relevant to spinlock contention.
  - Some Memscale features aim to improve contention in other areas.
- ASE CPU Busy % fell as spinlock contention fell.
- Throughput generally rose as spinlock contention fell.
- Some few exceptions, discussed in Appendix B.

Findings (2)

Benchmark set 1 (500-1000ms waits):

- Using ASE config set A , baseline vs. Memscale groups A-D (TSX, SNAP, LLDC, LFB, IMRS, HCB):
  - Memscale increased throughput by 236% (on same configuration).
  - Memscale decreased system table spinlock contention from 85.8% to 28.5%.
  - Memscale decreased ASE CPU Busy % from 89.7% to 82.0%.
  - With further ASE tuning, Memscale reduced system table spinlock contention to 0%!
  - This would not be possible under ASE 15.7.
  - This is the primary business case for ASE 16.0, and for ASE 16.0.x Memscale.
- This was quick and clumsy testing though.
  - Enabled most of the Memscale features at the same time.
  - We needed to gauge whether some helped and some hurt.

Findings (3)

Benchmark set 2 (50ms waits):

- Using ASE config set C (further tuning after all set 1 benchmarks):
  - TSX improved every measure.
  - TSX+SNAP worsened every measure compared to TSX alone.
  - TSX+SNAP+LLDC improved every measure.
  - TSX+SNAP+LLDC+LFB made either no difference or slight worsening.
  - TSX+SNAP+LLDC+LFB+IMRS was a mixed bag, some better, some worse.
  - TSX+SNAP+LLDC+LFB+IMRS+HCB regained lost ground.

Findings (4)

The sweet spot for this workload seemed to be TSX+SNAP+LLDC.

- Some implications that TSX+LLDC (no SNAP) might be better.
- Recall also that SNAP has many restrictions on when it can be used at all.

Adding LFB+IMRS+HCB gave only small improvements over that combination.

- But these require bespoke tuning for every database in every ASE.
  – … small bang for the buck?
- Some implications that TSX+LLDC+IMRS+HCB (no SNAP or LFB) might be better.

More detailed numbers and analysis in Appendix B.

Recommendations

Adopt TSX and LLDC for all ASE 16.0 environments.

- Cheap; easy; either zero or minimal configuration.

If the restrictions of SNAP are acceptable, enable it, but accept that it may not be used.

Consider IMRS+HCB on an exceptions basis when above doesn’t meet business requirements.

Further testing per environment required before using LFB.

- Might just be that our synthetic benchmark didn’t hit the use cases.
- Be alert to monitoring that suggests LFB might help.

Delay implementing NVCache or MVCC.

- Requires dedicated hardware or major code changes.

For further investigation

We did not benchmark some combinations of Memscale features:

- TSX+LLDC (no SNAP).
- TSX+LLDC+IMRS+HCB (no SNAP or LFB).

We would like to repeat the benchmarks with more detailed ASE MDA monitoring.

- Very much like to see what monSpinlocks reports.
- More in-depth analysis of MDA counters generally.
  - Better for drilling-down than sp_sysmon. (Yes Jeff, I admit it.)

Repeat using ASE SP03 PL04.

- CR log indicates some fixes and improvements to relevant features.

Most important – repeat using real workloads captured and replayed with Workload Analyzer.

Thank you.

Contact information:

Joe Woodhouse
Principal Consultant
Prima Donna Consulting
joe.woodhouse@primadonnaconsulting.com

Prima Donna Consulting gratefully acknowledges:

Dobler Consulting (US)

SAP (US)

SAP NS2 (US)

PRiMA DONNA
CONSULTING

THE BEST RUN SAP

Appendix A – RHEL Huge Pages

Appendix A – Huge Pages – TLB (1)

CPUs need to translate virtual memory addresses to physical addresses in RAM.

- All modern CPUs try to cache as much of this as possible on-chip, using page-based mapping in the Translation Lookaside Buffers (TLB).
- If a RAM memory address is not in the TLB a new mapping must be generated from a virtual to physical address. This “TLB fault” is expensive (orders of magnitude slower than finding the address in TLB).

The default memory page size is 4Kb, so 256Gb of RAM needs 67,108,864 memory pages to address.

- Even the most powerful CPUs are limited to how large these TLBs can be, and cannot possibly fit 67M+ pages in TLB.

Most modern CPUs allow a larger memory page size. Most x86_64 CPUs can use 2Mb “huge pages”. 256Gb of RAM needs only 131,072 pages to address. These still won’t fully fit in the TLB but far more of the RAM addresses will compared to 4kb memory page size.

- Some CPUs support a 1Gb memory page size. 256Gb RAM needs only 256 pages to address, which will fit in the TLB easily.

Appendix A – Huge Pages – TLB (2)

So why not always use huge pages? Because this becomes the minimum amount of memory that can be allocated, causing more pressure on memory.

- A process that could fit in 800Kb of memory (= 200 x 4Kb pages) would need 400Mb (= 200 x 2Mb Huge Pages).

The Linux implementation of Huge Pages cannot be swapped, putting more pressure still on memory.

If Huge Pages are not available then ordinary pages must be used instead.

- There is significant overhead in having to fall back to a smaller page size.
- Even more if a process must move memory pages between page sizes.

There are two mechanisms available for huge pages in Linux: regular hugepages and Transparent Huge Pages (THP).

Appendix A – Huge Pages – Huge Pages vs. THP (1)

RHEL Huge Pages are inconvenient:

- Static, can only be changed with a host reboot.
- Must be configured in the kernel (by root). Cannot be configured by DBAs.
- If ASE memory requirements change then full reboot of box required.
- Sizing depends on RAM and on ASE memory allocation.

One size does not fit all:

- Some RHEL configuration management tools assume that one size does fit all.
- Therefore Huge Pages are particularly inconvenient at sites using those tools.

RHEL Huge Pages are disabled by default.

Appendix A – Huge Pages – Huge Pages vs. THP (2)

Linux kernel developers agreed Huge Pages were a hassle, and introduced THP.

These are handled “transparently” in the background:

- Dynamically allocating huge pages.
- Dynamically reallocating huge to regular and vice versa.
- Background daemon process to defrag memory and turn many regular pages into fewer huge pages.

Unfortunately …

- All background page changes are currently ***synchronous***.
- This kills performance for very large memory allocations… like SAP ASE.
- Almost all database vendors recommend THP be disabled for this reason.

RHEL THP are enabled by default.

Appendix A – Huge Pages – Are we using Huge Pages?

Check using cat /proc/meminfo:

```
$ cat /proc/meminfo | grep Huge
AnonHugePages: 0 kB
HugePages_Total: 996147
HugePages_Free: 78131
HugePages_Rsvd: 918016
HugePages_Surp: 0
Hugepagesize: 2048 kB
```

Appendix A – Huge Pages – Overhead of memory page size

```
$ cat /proc/meminfo | grep Table
[…]
PageTables:  626200 kB
[…]
```

Normal 4Kb pages

```
$ cat /proc/meminfo | grep Table
[…]
PageTables:  17388 kB
[…]
```

2Mb Huge Pages

A memory saving of 594Mb doesn’t seem like much on a 2Tb host…

- But what % of 611Mb fits on-die in CPU TLP, vs. what % of 16.9Mb?

Appendix A – Huge Pages – Setting Huge Pages (1)

To use 2Mb Huge Pages for ASE, at a high level:

- 1. Calculate number of 2Mb Huge Pages required for ASE.
- 2. Add some headroom.
- 3. Configure RHEL to use this many Huge Pages.
- 4. Configure RHEL to allow “sybase” login to use this many Huge Pages (twice).
- 5. Configure ASE to use only Huge Pages.

Appendix A – Huge Pages – Setting Huge Pages (2)

Step 1: Calculate number of 2Mb Huge Pages required for ASE.

- ASE “max memory” is in 2Kb pages (the configured/run value, not memory used).
- Divide by 1024.
- ASE max memory 1.75Tb = 939,524,096 x 2Kb pages.
- 939,524,096 / 1024 = 917,504 x 2Mb Huge Pages.

Step 2: Add some headroom.

- Add +1Gb for ASE expansion.
- 1Gb = 1,024Mb = 512 x 2Mb Huge Pages.
- So we want 917,504 + 512 = 918,016 x 2Mb Huge Pages.

Appendix A – Huge Pages – Setting Huge Pages (3)

Step 3: Configure RHEL to use this many Huge Pages.

- Easiest way is to edit /etc/sysctl.conf and reboot:

```
vm.nr_hugepages=918016
vm.hugetlb_shm_group=<sybase login GID>          # may not always be needed
```

Appendix A – Huge Pages – Setting Huge Pages (4)

Step 4: Configure RHEL to allow “sybase” login to use this many Huge Pages (twice).

- 918,016 x 2Mb Huge Pages = 1,880,096,768.
- Documentation says to edit /etc/security.conf… Don’t! All settings here are overwritten by any files in /etc/security/limits.d/ .
- Create new file /etc/security/limits.d/99999-sybase-memlock.conf:

```
sybase soft memlock 1880096768
sybase hard memlock 1880096768
```

- Best practice: grep for memlock in any other file in /etc/security/limits.d/ .
  - Could be set by group instead of by user.
- Also need to add to Sybase RUN_<SERVER> file before dataserver line:

```
ulimt -l 1880096768
```

Appendix A – Huge Pages – Setting Huge Pages (5)

Step 5: Configure ASE to use (only) Huge Pages.

- Set in either ASE .cfg file or in ASE with sp_configure (needs reboot):

  “enable HugePages”=2
- Watch out: case sensitive parameter name!

Appendix A – Huge Pages – 1Gb Huge Pages

Some hardware also supports (“Extra”) Huge Pages of 1Gb.

To check:

```
$ egrep “pse|pdp1gb” /proc/cpuinfo | uniq
flags : [...] pse [...]
```

supports 2Mb Huge Pages

```
$ egrep “pse|pdp1gb” /proc/cpuinfo | uniq
flags : [...] pdpe1gb [...]
```

supports 1Gb Huge Pages

Standard Intel documentation does not show whether a particular CPU supports 1Gb “extra” Huge Pages.

Requires explicit vendor request and explicit vendor assurance. Sorry.

Setting 1Gb Huge Pages is best done in the kernel boot command line (via grub):

```
hugepagesz=1G hugepages=1793
```

Appendix A – Huge Pages – Are we using THP?

Check using cat /proc/meminfo:

```
$ cat /proc/meminfo | grep Huge
AnonHugePages: 34986 kB
HugePages_Total: 0
HugePages_Free: 0
HugePages_Rsvd: 0
HugePages_Surp: 0
Hugepagesize: 4 kB
```

Appendix A – Huge Pages – Setting THP

Most RHEL documentation says:

```
echo never >/sys/kernel/mm/transparent_hugepage/enabled
echo never >/sys/kernel/mm/transparent_hugepage/defrag
```

Unfortunately this does not seem to work.

- Trivial to prove that THP still enabled and used via “cat /proc/meminfo”.

Only way to be sure it to put it into RHEL boot command line:

```
transparent_hugepage=never
```

Requires configuration via “grub”.

- Watch out that configuration management tools do not override this!
- Recommend adding this to ***every*** ASE host boot command line.

Appendix A – Huge Pages – What to measure

TLB / HugePages effects were expected to be subtle and hard to measure.

Simply measuring “wall clock time” was unlikely to tell the full story.

- Particularly as all benchmarking was a synthetic workload intended to provoke TLB issues.

We had to look at many low-level events:

▪ TLB loads vs. misses	- TLB hit or miss
▪ CPU instructions executed per CPU clock cycle	- CPU effectiveness
▪ Memory PageTable cache misses	- memory map effectiveness
▪ Main memory reads per second	- memory effectiveness

Some of these are directly measurable; some must be derived from other measurements.

Appendix A – Huge Pages – How to measure (1)

Intel allows TLB events to be tracked by O/S via CPU and RAM microcode counters.

RHEL exposes microcode counters through “perf stat”.

The most common/important counters have human-meaningful aliases.

- However many of the ones we had to measure require “event IDs”.

Event IDs are platform-dependent hex codes.

- Platform-dependent here means “CPU architecture”.
- E.g. “Intel Microarchitecture Code Named Haswell”.

To the documentation!

Appendix A – Huge Pages – How to measure (2)

Step 1: What CPU are we using?

```
$ cat /proc/cpuinfo | grep ‘model name’
model name : Intel(R) Xeon(TM) E7-4850 CPU @ 2.10GHz
```

Step 2: What CPU code name (or “family”) is this?

- https://ark.intel.com/#@Processors

Code Name	Products formerly Broadwell

Step 3: Get the list of Intel PerfMon microcode counters for this CPU code name.

- https://download.01.org/perfmon/index/

Step 4: Look up the event microcodes.

- https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html

Appendix A – Huge Pages – How to measure (3)

RHEL “perf stat” already aliases some of what we need:

- dTLB-loads, dTLB-load-misses, dTLB-store-misses, iTLB-load-misses, instructions, cycles, cache-misses

```
e.g. perf stat –e instructions
```

We need microcodes for:

- d-cycles-waited, i-cycles-waited, d-main-memory-reads, i-main-memory-reads
- Events given in terms of a “umask” and an “event ID”.

```
e.g. perf stat -e cpu/event=0x08,umask=0x10,name=d-cycles-waited/
```

(All of the above documented at previous links.)

Actual perf stat statement used on next slide.

Appendix A – Huge Pages – How to measure (4)

```
perf stat \
  -a -S -B \
  -e dTLB-loads \
  -e dTLB-load-misses \
  -e dTLB-store-misses \
  -e iTLB-load-misses \
  -e instructions \
  -e cycles \
  -e cpu/event=0x08,umask=0x10,name=d-cycles-waited/ \
  -e cpu/event=0x85,umask=0x10,name=i-cycles-waited/ \
  -e cache-misses \
  -e cpu/event=0xbc,umask=0x18,name=d-main-memory-reads/ \
  -e cpu/event=0xbc,umask=0x28,name=i-main-memory-reads/ \
  ${*} \
  2>&1 \
| tee ${DIR}/perf.stat.${NNN}
```

Appendix A – Huge Pages – Detailed analysis (1)

	No THP, no HP	THP only	HP 2Mb	HP 1Gb
dTLB-loads	4,449,356,965,642,820	4,553,304,235,447,700	4,814,399,196,686,040	4,634,719,330,467,360
dTLB-load-misses	2,105,010,276,841	2,722,591,971,136	12,354,334,314,344	3,417,278,235,524
dTLB-store-misses	105,583,416,023	252,046,050,550	1,515,046,934,071	351,076,696,445
iTLB-load-misses	932,006,148,259	975,893,002,993	1,020,513,331,019	958,443,417,772
total TLB misses	0.071%	0.087%	0.309%	0.102%
instructions	12,531,110,754,255,500	12,829,673,814,421,200	13,562,335,014,020,300	13,063,587,096,099,100
cycles	11,409,758,430,713,700	11,740,823,613,125,800	12,503,603,923,361,300	11,887,027,487,599,800
instructions per cycle	1.098	1.093	1.085	1.099
d-cycles-waited	49,275,391,837,172	62,683,320,547,878	234,920,785,329,693	72,439,020,232,381
i-cycles-waited	20,529,010,929,301	22,414,746,483,985	28,033,736,336,195	22,182,344,194,742
total cycles waited	0.612%	0.725%	2.103%	0.796%
cache-misses	651,975,222,175	835,033,994,295	763,814,637,489	721,330,941,906
d-main-memory-reads	15,765,631,111	17,659,238,946	12,709,750,059	10,745,644,030
i-main-memory-reads	3,321,808,093	4,346,827,666	4,325,558,189	3,571,271,250
elapsed time (s)	42,870	41,240	42,414	40,461
total main memory reads / s	445,243.64	533,606.14	401,643.52	353,844.82

Appendix A – Huge Pages – Detailed analysis (2)

	No THP, no HP	THP only	HP 2Mb	HP 1Gb
dTLB-loads	4,449,356,965,642,820	4,553,304,235,447,700	4,814,399,196,686,040	4,634,719,330,467,360
dTLB-load-misses	2,105,010,276,841	2,722,591,971,136	12,354,334,314,344	3,417,278,235,524
dTLB-store-misses	105,583,416,023	252,046,050,550	1,515,046,934,071	351,076,696,445
iTLB-load-misses	932,006,148,259	975,893,002,993	1,020,513,331,019	958,443,417,772
total TLB misses	0.071%	0.087%	0.309%	0.102%
instructions	12,531,110,754,255,500	12,829,673,814,421,200	13,562,335,014,020,300	13,063,587,096,099,100
cycles	11,409,758,430,713,700	11,740,823,613,125,800	12,503,603,923,361,300	11,887,027,487,599,800
instructions per cycle	1.098	1.093	1.085	1.099
d-cycles-waited	49,275,391,837,172	62,683,320,547,878	234,920,785,329,693	72,439,020,232,381
i-cycles-waited	20,529,010,929,301	22,414,746,483,985	28,033,736,336,195	22,182,344,194,742
total cycles waited	0.612%	0.725%	2.103%	0.796%
cache-misses	651,975,222,175	835,033,994,295	763,814,637,489	721,330,941,906
d-main-memory-reads	15,765,631,111	17,659,238,946	12,709,750,059	10,745,644,030
i-main-memory-reads	3,321,808,093	4,346,827,666	4,325,558,189	3,571,271,250
elapsed time (s)	42,870	41,240	42,414	40,461
total main memory reads / s	445,243.64	533,606.14	401,643.52	353,844.82

Wall clock time makes THP look better than no THP or HugePages of standard 2Mb size…

Appendix A – Huge Pages – Detailed analysis (3)

	No THP, no HP	THP only	HP 2Mb	HP 1Gb
dTLB-loads	4,449,356,965,642,820	4,553,304,235,447,700	4,814,399,196,686,040	4,634,719,330,467,360
dTLB-load-misses	2,105,010,276,841	2,722,591,971,136	12,354,334,314,344	3,417,278,235,524
dTLB-store-misses	105,583,416,023	252,046,050,550	1,515,046,934,071	351,076,696,445
iTLB-load-misses	932,006,148,259	975,893,002,993	1,020,513,331,019	958,443,417,772
total TLB misses	0.071%	0.087%	0.309%	0.102%
instructions	12,531,110,754,255,500	12,829,673,814,421,200	13,562,335,014,020,300	13,063,587,096,099,100
cycles	11,409,758,430,713,700	11,740,823,613,125,800	12,503,603,923,361,300	11,887,027,487,599,800
instructions per cycle	1.098	1.093	1.085	1.099
d-cycles-waited	49,275,391,837,172	62,683,320,547,878	234,920,785,329,693	72,439,020,232,381
i-cycles-waited	20,529,010,929,301	22,414,746,483,985	28,033,736,336,195	22,182,344,194,742
total cycles waited	0.612%	0.725%	2.103%	0.796%
cache-misses	651,975,222,175	835,033,994,295	763,814,637,489	721,330,941,906
d-main-memory-reads	15,765,631,111	17,659,238,946	12,709,750,059	10,745,644,030
i-main-memory-reads	3,321,808,093	4,346,827,666	4,325,558,189	3,571,271,250
elapsed time (s)	42,870	41,240	42,414	40,461
total main memory reads / s	445,243.64	533,606.14	401,643.52	353,844.82

… but we see the best CPU and memory effectiveness with 1Gb HugePages

Appendix A – Huge Pages – Detailed analysis (4)

	No THP, no HP	THP only	HP 2Mb	HP 1Gb
dTLB-loads	4,449,356,965,642,820	4,553,304,235,447,700	4,814,399,196,686,040	4,634,719,330,467,360
dTLB-load-misses	2,105,010,276,841	2,722,591,971,136	12,354,334,314,344	3,417,278,235,524
dTLB-store-misses	105,583,416,023	252,046,050,550	1,515,046,934,071	351,076,696,445
iTLB-load-misses	932,006,148,259	975,893,002,993	1,020,513,331,019	958,443,417,772
total TLB misses	0.071%	0.087%	0.309%	0.102%
instructions	12,531,110,754,255,500	12,829,673,814,421,200	13,562,335,014,020,300	13,063,587,096,099,100
cycles	11,409,758,430,713,700	11,740,823,613,125,800	12,503,603,923,361,300	11,887,027,487,599,800
instructions per cycle	1.098	1.093	1.085	1.099
d-cycles-waited	49,275,391,837,172	62,683,320,547,878	234,920,785,329,693	72,439,020,232,381
i-cycles-waited	20,529,010,929,301	22,414,746,483,985	28,033,736,336,195	22,182,344,194,742
total cycles waited	0.612%	0.725%	2.103%	0.796%
cache-misses	651,975,222,175	835,033,994,295	763,814,637,489	721,330,941,906
d-main-memory-reads	15,765,631,111	17,659,238,946	12,709,750,059	10,745,644,030
i-main-memory-reads	3,321,808,093	4,346,827,666	4,325,558,189	3,571,271,250
elapsed time (s)	42,870	41,240	42,414	40,461
total main memory reads / s	445,243.64	533,606.14	401,643.52	353,844.82

2Mb Huge Pages were worse than baseline in almost every measure, so why was it slightly faster?

Appendix A – Huge Pages – Detailed analysis (5)

	No THP, no HP	THP only	HP 2Mb	HP 1Gb
dTLB-loads	4,449,356,965,642,820	4,553,304,235,447,700	4,814,399,196,686,040	4,634,719,330,467,360
dTLB-load-misses	2,105,010,276,841	2,722,591,971,136	12,354,334,314,344	3,417,278,235,524
dTLB-store-misses	105,583,416,023	252,046,050,550	1,515,046,934,071	351,076,696,445
iTLB-load-misses	932,006,148,259	975,893,002,993	1,020,513,331,019	958,443,417,772
total TLB misses	0.071%	0.087%	0.309%	0.102%
instructions	12,531,110,754,255,500	12,829,673,814,421,200	13,562,335,014,020,300	13,063,587,096,099,100
cycles	11,409,758,430,713,700	11,740,823,613,125,800	12,503,603,923,361,300	11,887,027,487,599,800
instructions per cycle	1.098	1.093	1.085	1.099
d-cycles-waited	49,275,391,837,172	62,683,320,547,878	234,920,785,329,693	72,439,020,232,381
i-cycles-waited	20,529,010,929,301	22,414,746,483,985	28,033,736,336,195	22,182,344,194,742
total cycles waited	0.612%	0.725%	2.103%	0.796%
cache-misses	651,975,222,175	835,033,994,295	763,814,637,489	721,330,941,906
d-main-memory-reads	15,765,631,111	17,659,238,946	12,709,750,059	10,745,644,030
i-main-memory-reads	3,321,808,093	4,346,827,666	4,325,558,189	3,571,271,250
elapsed time (s)	42,870	41,240	42,414	40,461
total main memory reads / s	445,243.64	533,606.14	401,643.52	353,844.82

… Because larger memory page table meant we went to main memory less, and that adds up.

Appendix A – Huge Pages – Detailed analysis (6)

	No THP, no HP	THP only	HP 2Mb	HP 1Gb
dTLB-loads	4,449,356,965,642,820	4,553,304,235,447,700	4,814,399,196,686,040	4,634,719,330,467,360
dTLB-load-misses	2,105,010,276,841	2,722,591,971,136	12,354,334,314,344	3,417,278,235,524
dTLB-store-misses	105,583,416,023	252,046,050,550	1,515,046,934,071	351,076,696,445
iTLB-load-misses	932,006,148,259	975,893,002,993	1,020,513,331,019	958,443,417,772
total TLB misses	0.071%	0.087%	0.309%	0.102%
instructions	12,531,110,754,255,500	12,829,673,814,421,200	13,562,335,014,020,300	13,063,587,096,099,100
cycles	11,409,758,430,713,700	11,740,823,613,125,800	12,503,603,923,361,300	11,887,027,487,599,800
instructions per cycle	1.098	1.093	1.085	1.099
d-cycles-waited	49,275,391,837,172	62,683,320,547,878	234,920,785,329,693	72,439,020,232,381
i-cycles-waited	20,529,010,929,301	22,414,746,483,985	28,033,736,336,195	22,182,344,194,742
total cycles waited	0.612%	0.725%	2.103%	0.796%
cache-misses	651,975,222,175	835,033,994,295	763,814,637,489	721,330,941,906
d-main-memory-reads	15,765,631,111	17,659,238,946	12,709,750,059	10,745,644,030
i-main-memory-reads	3,321,808,093	4,346,827,666	4,325,558,189	3,571,271,250
elapsed time (s)	42,870	41,240	42,414	40,461
total main memory reads / s	445,243.64	533,606.14	401,643.52	353,844.82

Also notice that on all of these measures, 1Gb Huge Pages further outperformed 2Mb Huge Pages. This is why 1Gb was the best even though on some less important measures it didn't do as well.

Appendix A – Huge Pages – Detailed analysis (7)

What can we conclude from these measurements?

Even though main memory accesses are fast (ns), they add up.

- RAM access time is ~50ns… which doesn’t sound like much. But this is misleading.
- There are precharge delays and latencies between accesses.
- RAM cycle time is more like 100-120ns. That still doesn’t sound like much…
- … but it is probably ~10 CPU clock cycles.
- CPU is still at least an order of magnitude faster than RAM.
- So CPU blocked waiting on RAM hurts performance a lot.

Appendix A – Huge Pages – Detailed analysis (8)

So, dropping main memory accesses from ~450,000/s to ~350,000/s:

- Saved 100,000 x 100ns = 0.01s per second.
- … and more importantly has saved 100,000 x 10 clock cycles = 1M CPU clock cycles per second!

Put another way, because CPU didn’t have to wait on memory quite so much, it was able to spend 1M cycles per second on other tasks.

Lesson learned: main memory access is fast, but CPU still waits on memory.

- Removing 1% of wall clock waits counts for some.
- Removing one million CPU waits per second counts for a ***lot***.
- … Even if some other measures needed to be higher to achieve this.

Appendix B – ASE Memscale

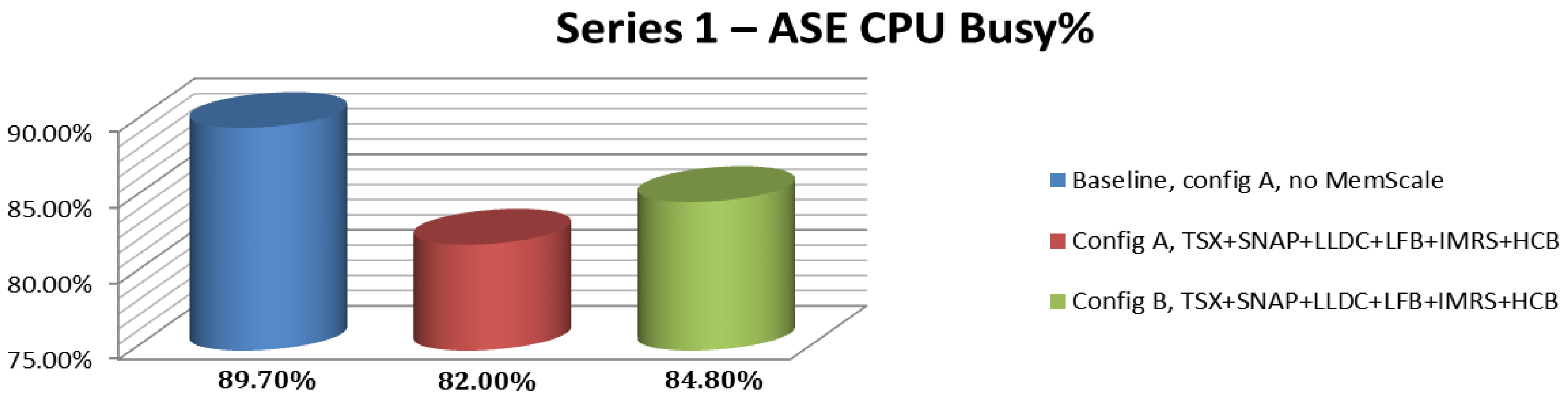
Appendix B – Memscale – Benchmarks (1)

Benchmark		500-1000ms waits			Spinlock Contention		
Series	Run	Description	ASE CPU Busy%	xacts per ASE CPU unit	default data cache	system tables cache	log cache
1	03	Baseline, config A, no Memscale	89.70%	97.3	2.10%	85.80%	2.30%
1	02	Config A, TSX+SNAP+LLDC+LFB+IMRS+HCB	82.00%	230.1	66.00%	28.50%	10.40%
1	04	Config B, TSX+SNAP+LLDC+LFB+IMRS+HCB	84.80%	198.6	66.70%	0.00%	10.80%

lower is better

higher is better

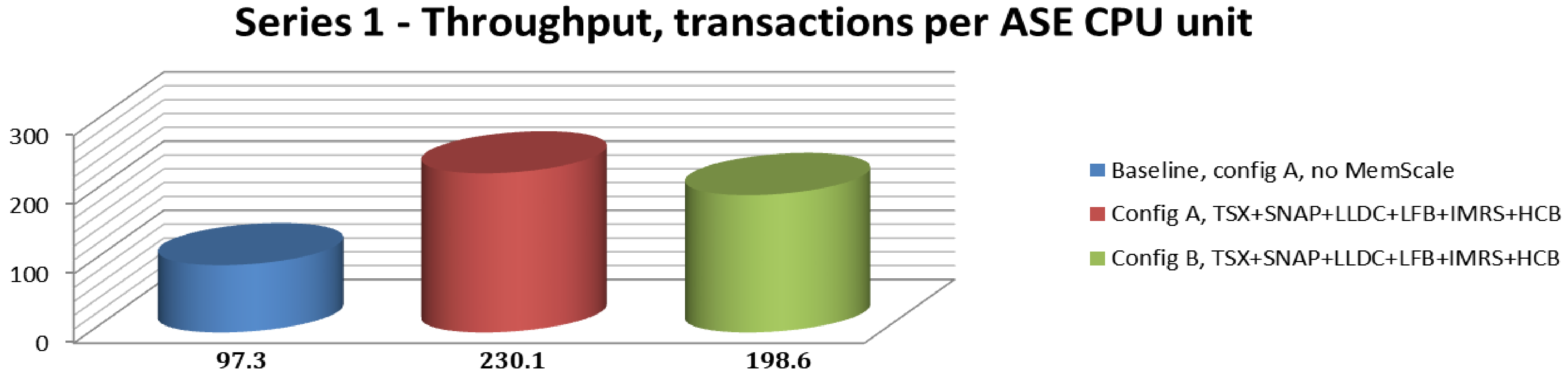
Appendix B – Memscale – Benchmarks (2)



As expected, Memscale reduced CPU Busy %. (Lower is better).

- Further ASE tuning (config set B) improved other measures but worsened CPU Busy %.
- Tentative explanation: was now able to get more work done?

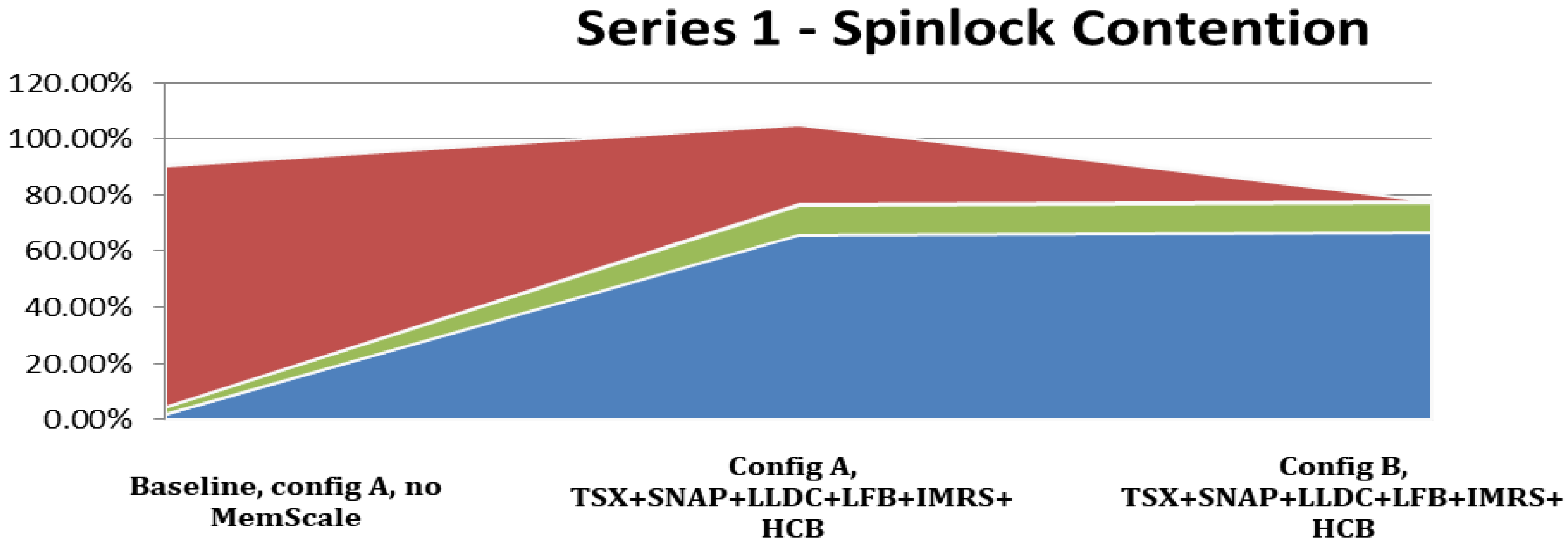
Appendix B – Memscale – Benchmarks (3)



As expected, Memscale increased throughput (Higher is better).

- Further ASE tuning (config set B) improved other measures but worsened throughput.
- Currently no good explanation! Would like to monitor in more detail.

Appendix B – Memscale – Benchmarks (4)



As expected, Memscale decreased spinlock contention overall. (Lower is better).

- Until Memscale, the bottleneck was the system tables cache.
- Memscale but ASE tuning set B brought system tables cache spinlock contention to zero.
  - But increased spinlock contention elsewhere, especially in default data cache.
  - Total spinlock contention fell.

Appendix B – Memscale – Benchmarks (5)

It is the nature of performance tuning to uncover new bottlenecks as each bottleneck is fixed.

We never knew what the next bottleneck was because we never got that far until now.

Easing the #1 area of spinlock contention exposed the #2 area.

- And gave us a bit of a clue about a likely later bottleneck.

What this told us is that we needed:

- More sophisticated ASE tuning to deal with this.
  - We evolved ASE tuning set C in response.
- More sophisticated benchmark design.
  - Up until now we were “all or nothing”.
  - We now wondered if some Memscale features helped while others hurt.
  - We also wanted more nuance in how & where we enabled each Memscale feature.

Appendix B – Memscale – Further benchmarks (1)

New benchmark series 2 addressed these issues.

ASE tuning set C evolved in response to apparent regression in some measures with tuning set B.

- Also specifically designed to reduced the new bottlenecks discovered in series 1.
- Enable one Memscale feature at a time.
  - In order of “feasibility/difficulty” groups based on ease of implementation.
- Used Memscale features more intelligently.
  - i.e. didn’t blindly follow synthetic trading simulation test guidelines.
  - e.g. heavy spinlock contention in default data cache?
  - Once LLDC enabled, try LLDC in default data cache too.

Lastly we wanted to see if we could stress ASE even further.

- Wait times between trades & orders reduced from 1000ms or 500ms to 50ms.
  - This is no longer a realistic benchmark, but a stress test.

Appendix B – Memscale – Further benchmarks (2)

Benchmark		50ms waits			Spinlock Contention		
Series	Run	Description	ASE CPU Busy%	xacts per ASE CPU unit	default data cache	system tables cache	log cache
2	05	Baseline, reduced waits, config C, no Memscale	89.40%	91.1	2.10%	85.60%	4.10%
2	12	Reduced waits, config C, TSX	86.30%	108.6	4.00%	57.90%	3.10%
2	13	Reduced waits, config C, TSX+SNAP	86.70%	102.6	37.50%	91.60%	10.70%
2	14	Reduced waits, config C, TSX+SNAP+LLDC	82.70%	155.8	0.40%	0.00%	12.30%
2	15	Reduced waits, config C, TSX+SNAP+LLDC+LFB	82.20%	152.2	0.40%	0.00%	12.30%
2	07	Reduced waits, config C, TSX+SNAP+LLDC+LFB+IMRS	81.70%	124.1	0.20%	0.00%	15.90%
2	06	Reduced waits, config C, TSX+SNAP+LLDC+LFB+IMRS+HCB	82.20%	159.7	0.30%	0.00%	12.80%

lower is better

higher is better

Note: benchmarks run IDs are out of order due to repeating some benchmarks

Appendix B – Memscale – Further benchmarks (3)

Notice apparent regressions when enabling SNAP, LFB, and IMRS.

- Unclear if this was just how we implemented them, or artefacts of our synthetic benchmark.
- We didn't test SNAP on its own, nor LFB, nor IMRS.

SNAP regression was a surprise.

- This was expected to be a no-brainer to implement, and a cheap & easy win.
- Very much wanted to test this with real workload.

LFB and IMRS regression doesn't necessarily rule them out.

- They are expected to have some overhead, particularly IMRS.
- If they weren't fixing the then #1 bottleneck, that overhead wasn't offset by a gain.
- These still have a place in the ASE tuning toolbox.

Appendix B – Memscale – Further benchmarks (4)

Notice apparent sweet spots.

TSX+SNAP+LLDC+LFB+IMRS+HCB was the overall winner.

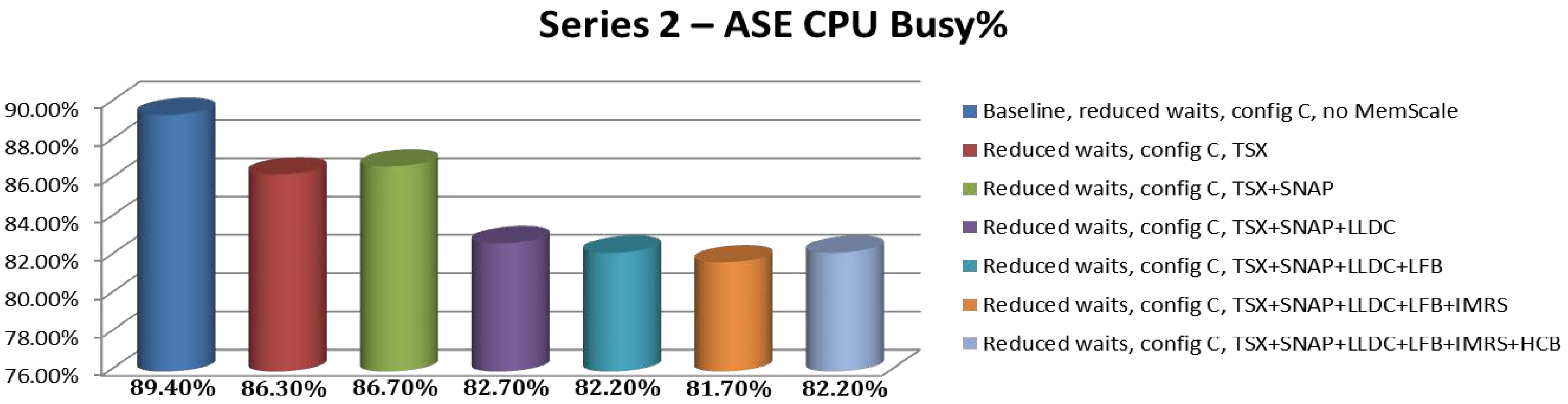
- But offered only very small gains over TSX+SNAP+LLDC.
  - Which is much easier to implement.
  - LFB needs some monitoring & analysis to setup correctly.
  - IMRS & HCB need a lot of configuration and tuning.

So the smart money says just go with TSX+SNAP+LLDC.

But remember previous apparent regression of SNAP?

- Is it possible that TSX+LLDC is even better?
- Or for that matter, maybe TSX+LLDC+IMRS+HCB?
- For further investigation…

Appendix B – Memscale – Analysis (1)

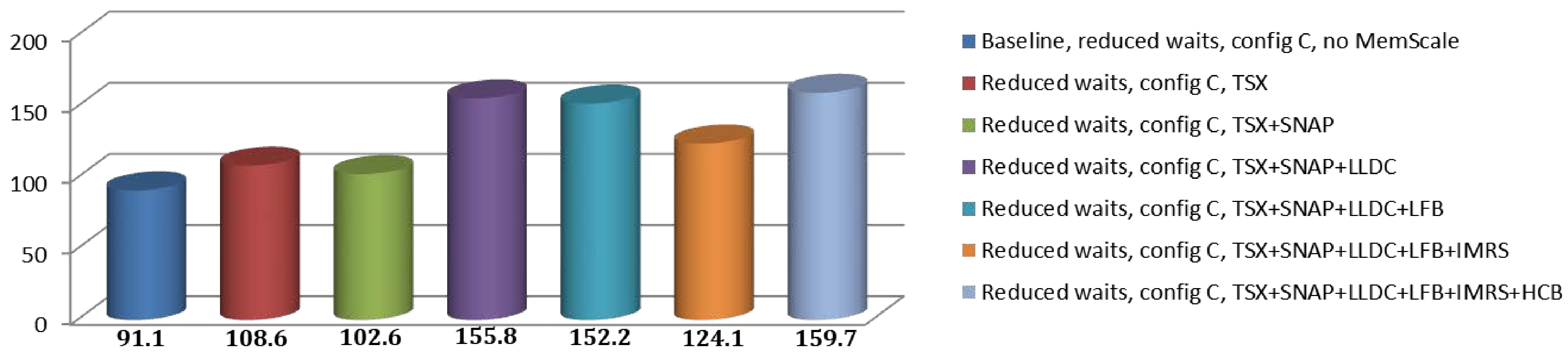


As expected, Memscale generally decreased CPU Busy %. (Lower is better).

- Some small apparent regressions with SNAP, LFB, and IMRS (without HCB).
- This makes sense if those measures weren’t improving whatever the #1 bottleneck was.
- These might still be useful.

Appendix B – Memscale – Analysis (2)

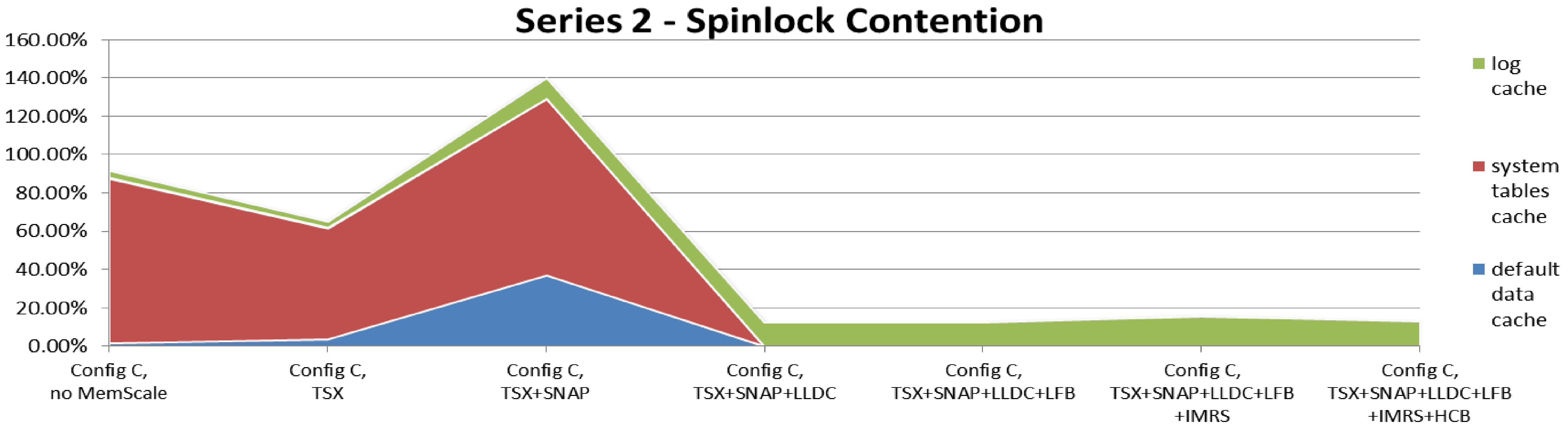
Series 2 – Throughput, transactions per ASE CPU unit



As expected, Memscale generally increased throughput. (Higher is better).

- Exactly the same apparent regressions with SNAP, LFB, and IMRS (without HCB).
- This still makes sense if those measures weren’t improving the #1 bottleneck.
- These might still be useful.

Appendix B – Memscale – Analysis (3)



Regression pattern looks very different! What’s happening here?

Appendix B – Memscale – Analysis (4)

Spinlock contention graph looks very different to the others.

Only regression observed was adding SNAP to TSX.

- This actually makes sense.
- SNAP is a Memscale feature to make SQL code execute more quickly.
- It does absolutely nothing for spinlock contention.
- What happens to spinlock contention when things are run faster?
  – Spinlock contention gets worse! This is exactly what we saw.

Series 2 spinlock contention graph also looks very different to series 1.

- We reduced waits to an absurdly low 50ms, but spinlock contention is a lot better.
- Why? We were smarter in how we used Memscale features.
- In particular, we made the default data cache LLDC.
- Series 1 benchmarks did not do this.

Appendix B – Memscale – Analysis (5)

As with other measures, the sweet spots are still the same:

- TSX+SNAP+LLDC, and TSX+SNAP+LLDC+LFB+IMRS+HCB.
  - The first combination is very easy to implement in all ASE 16 environments.
- Further benchmarking should test some of the remaining combinations.
  - TSX+LLDC.
  - TSX+LLDC+SNAP.
  - TSX+LLDC+IMRS+HCB.

This all makes sense as LFB, IMRS and HCB are all exquisitely sensitive to workload and schema.

It might just be that our synthetic benchmark did not sufficiently stress areas of ASE that benefit from those Memscale features.

This highlights the business value of benchmarking against genuine business workload.