Advances in Virtualization In Support of In-Memory Big Data Applications

TidalScale

SCALE | SIMPLIFY | OPTIMIZE | EVOLVE

Ike Nassi
Ike.nassi@tidalscale.com

What is the Problem We Solve?

Fast access to “big data”

- Faster access if data is “in memory”
- But often there’s too much data – can’t always fit it in memory
- Can’t always buy a bigger computer (with more memory)

So, the only alternative has been to “scale out”

- Generally requires new software or a software re-write ($$)
- Scale-out is not always easy

Solution is to create a single larger virtual machine

- Allows users to scale-up without $$ supercomputer HW costs
- No software changes required
- ***And, we’ll show you how well this works!***

Status

Up and running for 11 months

Compatible with all software tested

Automated 24x7 test system

3 customer trials completed in the last 6 weeks (Analytics, Big Data, EDA)

Supporting Centos 6.5, 6.6, 6.7, RHEL, FreeBSD

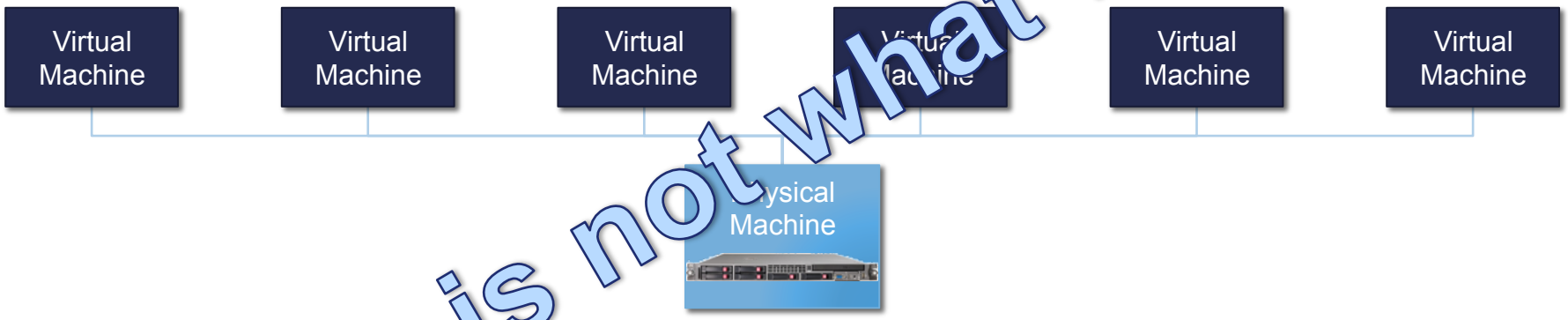
Passing full suite of Linux Test Project tests (for servers)

Preparing RC-2

25 node cluster for dev, 3(+1) x 5 node clusters for customers & test

Not a research project any longer

Traditional View of Virtualization

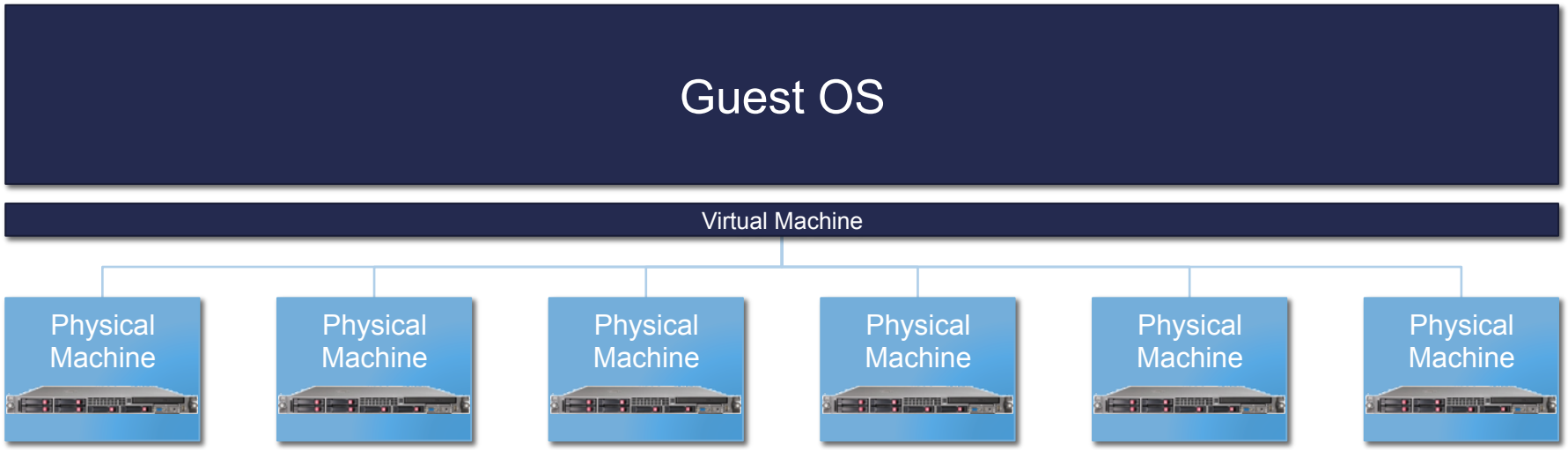


This is not what we do!

Virtual Machines sharing a server

TidalScale

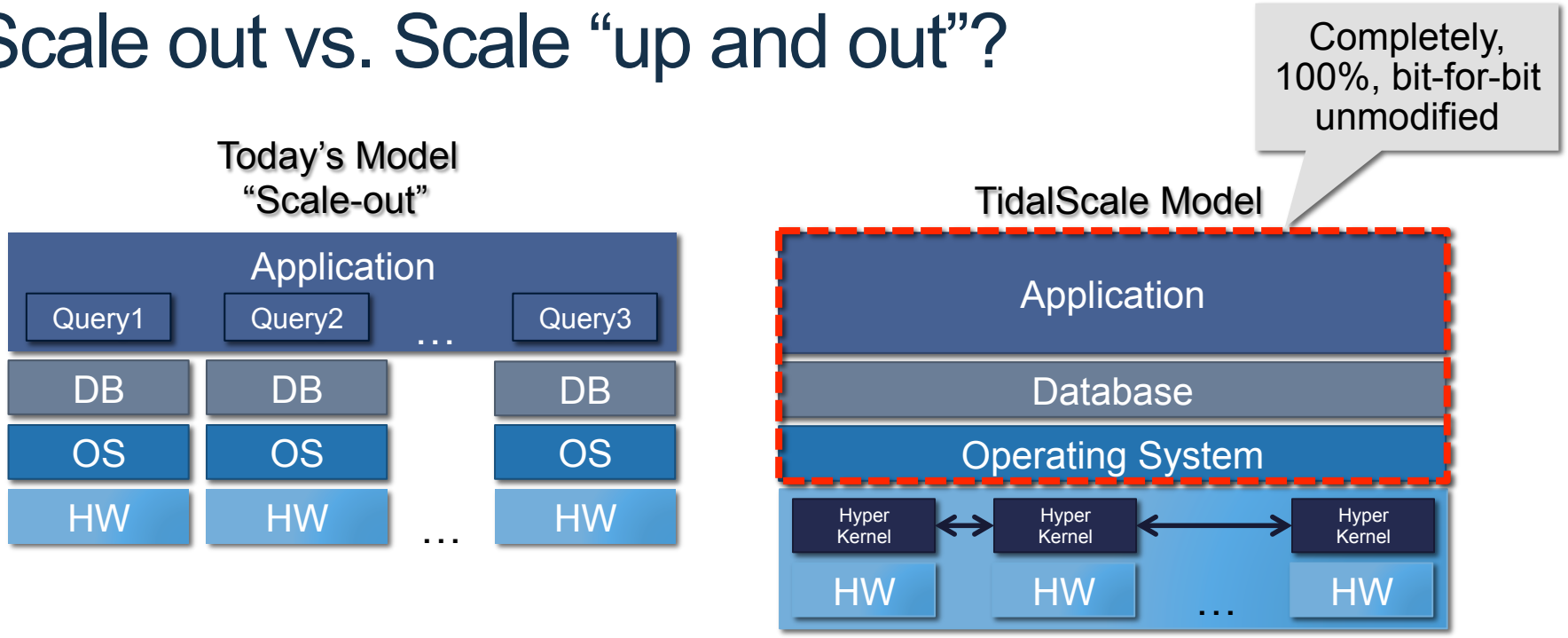
Alternative: Provide Hardware Aggregation



A Single Large Virtual Machine running on multiple servers (Inverse Virtualization)

TidalScale

Scale out vs. Scale “up and out”?



TidalScale

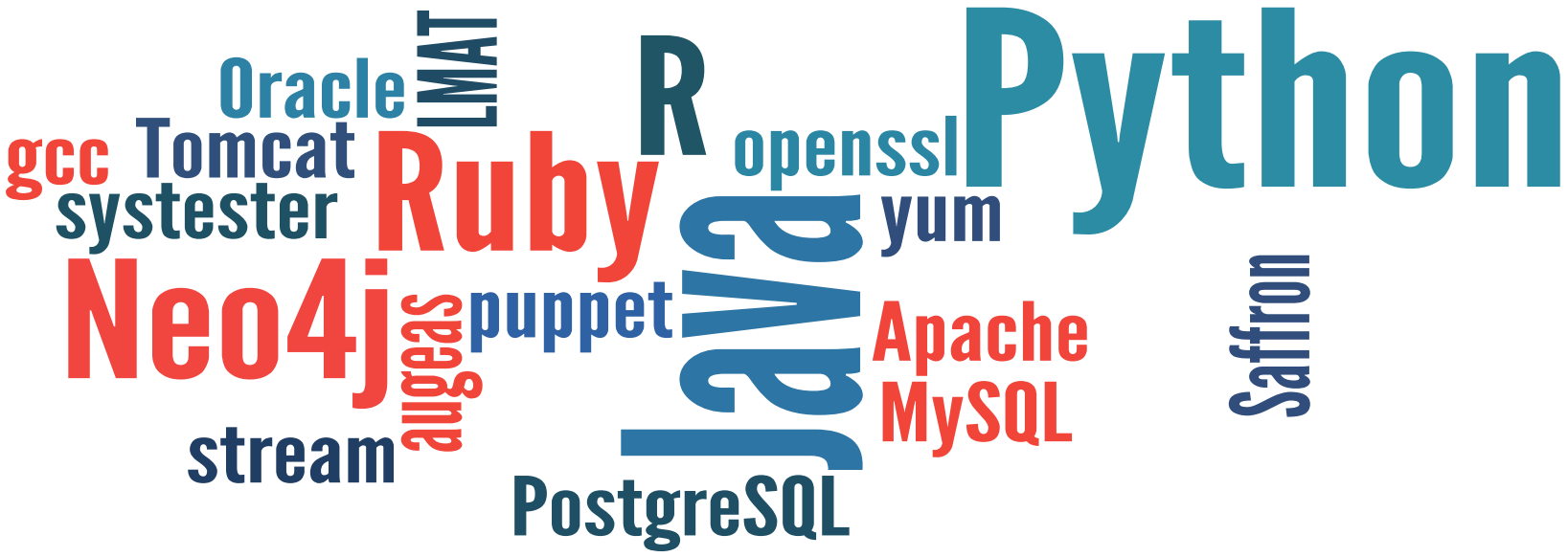
Scale up **and** Scale out (The Best of Both)

Software Simplicity	HW Cost
✔	✔

TidalScale

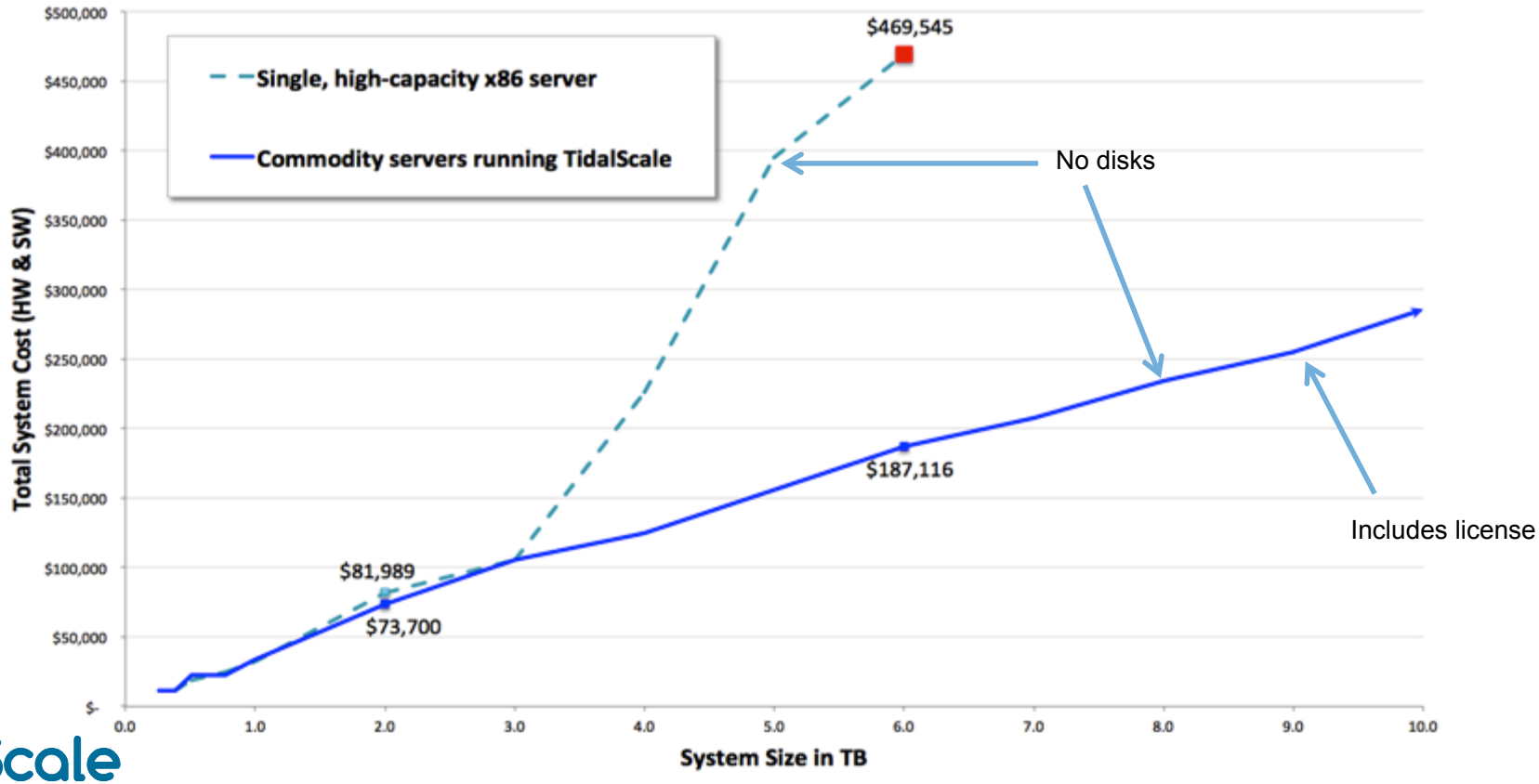
Everything Just Works

Just a few examples of many…



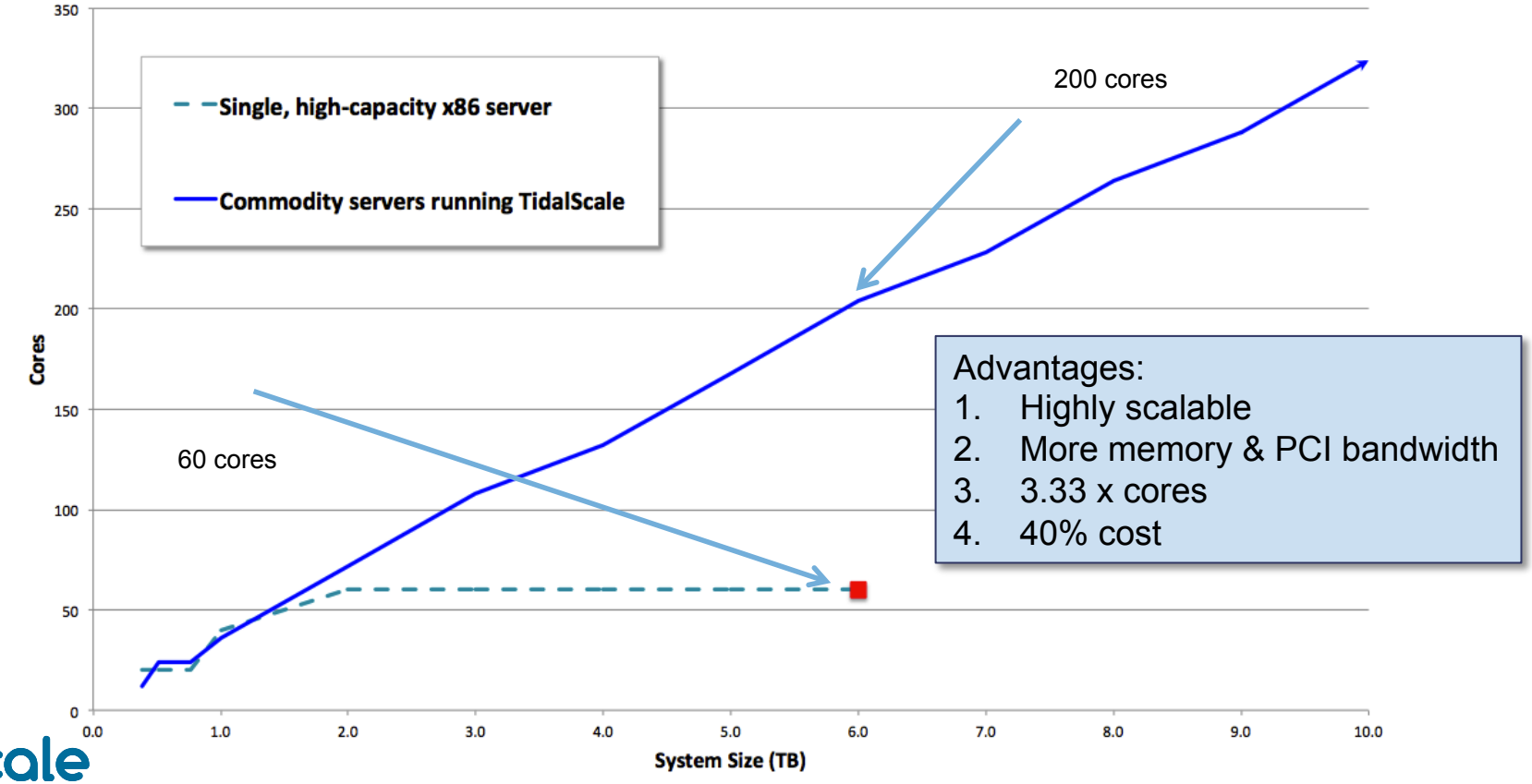
TidalScale

Price/Performance at Scale (late 2014, list price)



TidalScale

Performance Scales Up



TidalScale

Recent Hardware Example

POC-2 HW
120 processors @ 3.4GHz
3.84 TB
20x500GB SSD
< $ 80K (with tax)

As currently booted:
POC-2 current guest vm
3.2 TB
64 processors @ 3.4GHz
2x500GB SSD (>2 in test)

Screenshots

```
[tsadmin@poc-2 ~]$ free
             total       used       free     shared    buffers     cached
Mem:    3206066132   20635472 3185430660        156      45120     579628
-/+ buffers/cache:   20010724 3186055408
Swap:           0          0          0
[tsadmin@poc-2 ~]$ free -h
             total       used       free     shared    buffers     cached
Mem:          3.0T        19G       3.0T       156K        44M       566M
-/+ buffers/cache:        19G       3.0T
Swap:           0B         0B         0B
[tsadmin@poc-2 ~]$
```

3.2 TB DRAM

```
[tsadmin@poc-2 ~]$ cat /proc/cpuinfo|grep processor|wc -l
64
[tsadmin@poc-2 ~]$
```

64 processors

TidalScale

Working Sets*

45 years ago we figured out how to virtualize memory using the WS locality.

Today, locality is applied ubiquitously across our computing infrastructure.

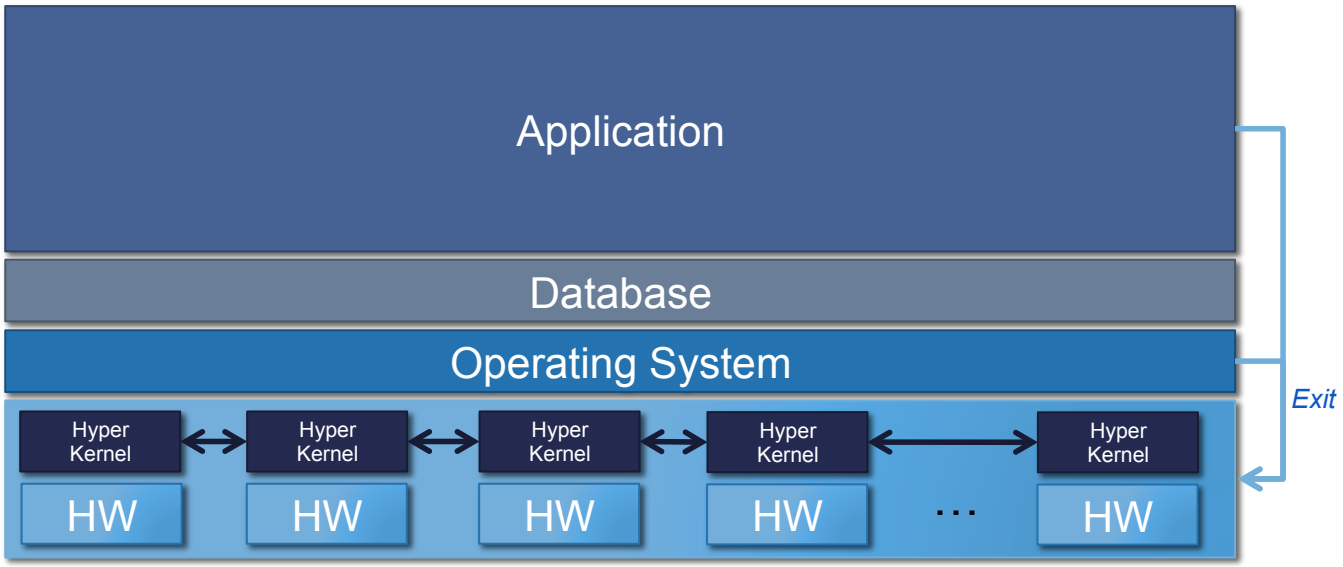
TidalScale applies *mobility & locality* to **all** primary resource types (processors, memory, Ethernets, storage) automatically & dynamically across physical machines.

* P.J. Denning

Nodes, Memory and Processors

- The hyperkernels aggregate all the processors, all the memory, all the storage, all the Ethernets (except for the private interconnect that we treat more as a system bus than a network).
- We treat these as *virtual and mobile*. The hyperkernels bind the virtual resources to physical resources, on demand. We move pages, vcpu's, interrupts, clocks, etc.
- There is *no master* node and *no shared state* among instances of the hyperkernel. Scheduling is purely distributed and peer-to-peer.
- The hyperkernel uses hardware virtualization extensions; the guest OS uses the first level page tables, the hyperkernels control the second level page tables.
- Memory is strongly coherent. We keep multiple read only copies of pages across the cluster, and do the normal invalidation when writes change page ownership.
- The hyperkernel sits beneath the OS and above the hardware. The VM looks like hardware to the OS and the hyperkernel does not need to know which OS or applications are running.

When Does the HK Get Involved?



TidalScale

Exits = Stalls

No exits = no overhead *[in which case we run at machine speed]*

The hyperkernel binds virtual processors to physical processors

Each exit is caused by an event that cannot be handled by the guest

e.g. remote interrupts, access to a remote I/O device, remote memory access, etc.

It's trapped and analyzed; strategies are evaluated

For each exit type, there is a unique set of strategies with costs:

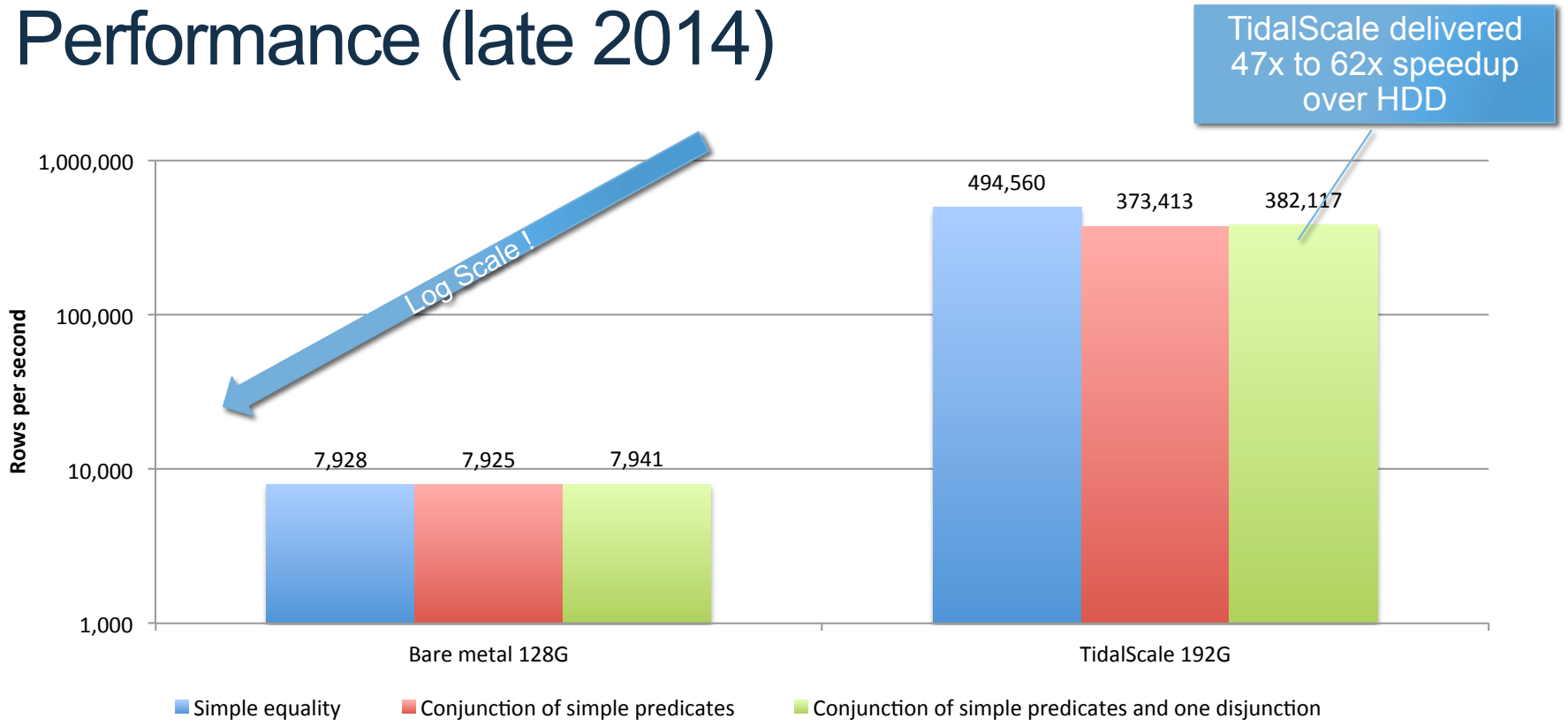
$$cost_{s1} = w_1 \bullet f_1^{i1} + w_2 \bullet f_2^{i2} + \cdots + w_n \bullet f_n^{in}$$

$$cost_{s2} = w_1 \bullet f_1^{i1} + w_2 \bullet f_2^{i2} + \cdots + w_n \bullet f_n^{in}$$

$cost_{selected} = min(costs^*)$ *[migrate a vcpu, or move/copy page]*

The hyperkernel keeps track of the updated state as we go along

Performance (late 2014)



- 3 query types across 100 Million rows
- Test performed with 4 cores at 2200 MHz
- TidalScale system was 2x96G

TidalScale

Performance (always a work in progress)

Our goal is to test in-memory performance on the machine described earlier: 5 nodes, 3.2TB, 64 processors, 2x 500GB SSD

TPC-H data by itself does not sufficiently exercise TS

So, we load up *mysqld* simultaneously with both TPC-H data *and* customer data

Our test varies memory configuration and compares performance with and without vCPU migration and/or page spillover

Experiment: Memory setup

Experiments	mysqld memory config	1	2	3
Memory		SSD	400GB, no migration	1 TB, with migration
innodb cache	1 TB	0	400 GB	400 GB
mysqld temp memory	1 TB/table 1.5 TB heap	0	0	600 GB

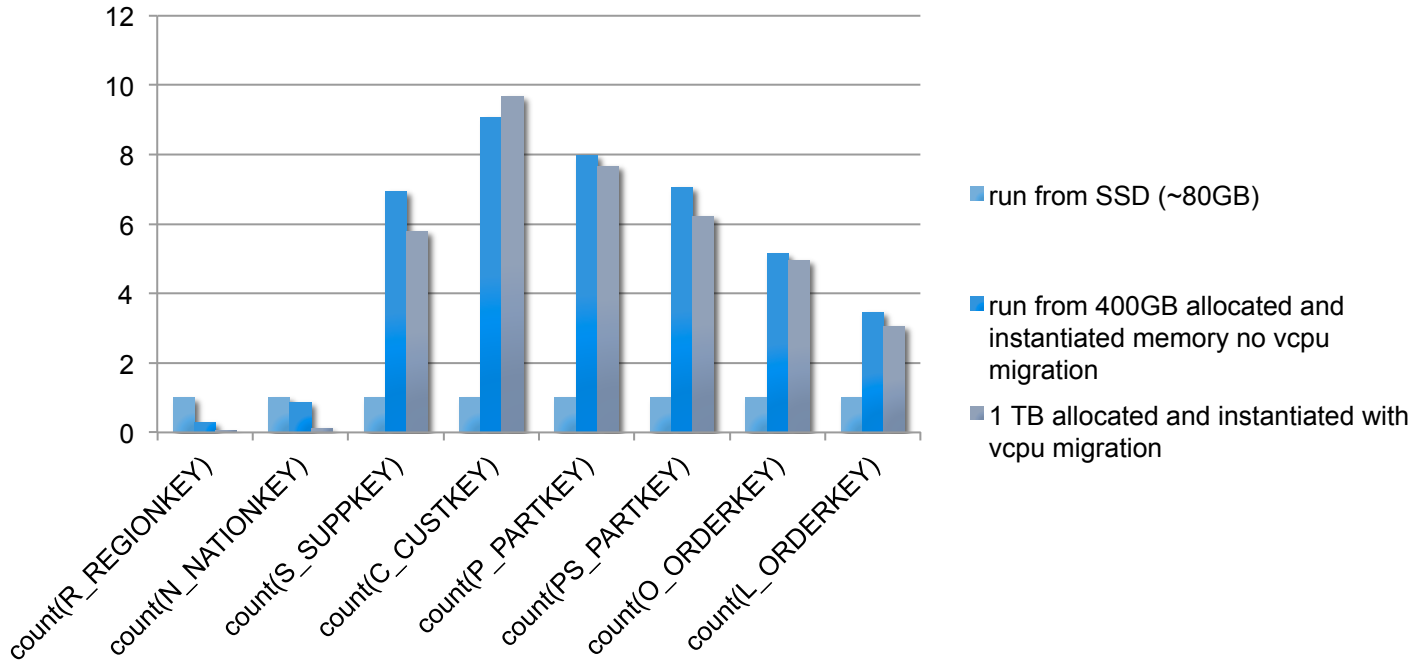
TidalScale

The experiment

```
i=tpch

echo `time sh -c "echo 'select count(C_CUSTKEY) from customer' | mysql -u tpch -ptpch $i"` &
echo `time sh -c "echo 'select count(P_PARTKEY) from part' | mysql -u tpch -ptpch $i"` &
echo `time sh -c "echo 'select count(N_NATIONKEY) from nation' | mysql -u tpch -ptpch $i"` &
echo `time sh -c "echo 'select count(R_REGIONKEY) from region' | mysql -u tpch -ptpch $i"` &
echo `time sh -c "echo 'select count(S_SUPPKEY) from supplier' | mysql -u tpch -ptpch $i"` &
echo `time sh -c "echo 'select count(PS_PARTKEY) from partsupp' | mysql -u tpch -ptpch $i"` &
echo `time sh -c "echo 'select count(O_ORDERKEY) from orders' | mysql -u tpch -ptpch $i"` &
echo `time sh -c "echo 'select count(L_ORDERKEY) from lineitem' | mysql -u tpch -ptpch $i"` &
wait
```

Customer + TPC-H (speedup)



TidalScale

Analysis of the experiment

1. Using larger memories gives us 3x – 9X performance speedup over already fast SSDs
2. Use your favorite factor to extrapolate to HDDs
3. While we processed 400GB of data into cache, we, in fact, allocated an additional 600GB (1TB - .4TB), so we had *a lot of memory cache headroom* on the existing hardware for processing larger data sets
4. The difference between experiment 2 and 3 is that you can now have 600GB of temporary tables *in memory* that can be used for subsequent work *without* having to write the temporary results to disk.
5. Migration *did not appreciably slow down* the elapsed time between experiment 2 and experiment 3.

Difficulty: How to evaluate performance

Try before you buy: currently we try to test real apps on real data on systems we host

If it doesn't crash, and it works with all tested software, then what?

Scalability based on data size?

- Linearity? Who knows? It's algorithm dependent

Fix the data size, scale the diameter?

- Better

Best might be to have constant data size, select *deterministic tests,* vary cluster diameter.

I'm open to other suggestions!

Lessons Learned - I

Keeping more data in memory reduces paging overhead

- Duh… but not so easy
- Gentle waves are better than “tidalwaves”
- Traditional HW has one memory wall - we have “multiple walls”

Hyperkernel minimizes or eliminates shared state

- Each node looks out for itself – increased recovery for mental breakdowns

Adhere to the *prime directive*:

- Look like hardware
- Never change the guest OS
- Never change the app

Intuition about the necessary speed of the interconnect is generally wrong. We don’t saturate the interconnect.

Lessons Learned - II

When things go wrong, system recovers

- Not always what you want when in development phase!

Train the system, rather than tune the system

Synchronizing distributed time is hard

Because we don't change the guest or the apps, our users don't risk breaking them

It's hard to introduce hyperkernel bugs when we're not modifying code

Scale the computer to the problem, not the converse

Don't necessarily believe people who say

- "It can't be done!" –or-
- "It's been tried before, and it doesn't work"

Summary

Scale:

- Aggregates compute resources for large scale in-memory analysis and decision support
- Scales like a cluster using commodity hardware, at linear cost
- Allows customers to grow gradually as their needs develop

Simplify:

- Dramatically simplifies application development: no changes!
- No need to distribute data or work across servers
- Existing applications run as a single instance, without modification, as if on a highly flexible mainframe

Optimize:

- Automatic dynamic hierarchical resource optimization

Evolve:

- Applicable to modern and emerging microprocessors, memories, interconnects, persistent storage & networks

TidalScale

SCALE | SIMPLIFY | OPTIMIZE | EVOLVE

Contact: Ike Nassi
ike.nassi@tidalscale.com