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

Virtual Machines sharing a server

This is not what we do!
Alternative: Provide Hardware Aggregation

A Single Large Virtual Machine running on multiple servers (Inverse Virtualization)
Scale out vs. Scale “up and out”?

Today’s Model
“Scale-out”

TidalScale Model

Completely, 100%, bit-for-bit unmodified
## Scale up and Scale out

(The Best of Both)

<table>
<thead>
<tr>
<th>Software Simplicity</th>
<th>HW Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
Everything Just Works

Just a few examples of many…
Price/Performance at Scale (late 2014, list price)

- Single, high-capacity x86 server
- Commodity servers running TidalScale

No disks
Includes license

Total System Cost (HW & SW)

System Size in TB

$73,700
$81,989
$187,116
$469,545
Performance Scales Up

Advantages:
1. Highly scalable
2. More memory & PCI bandwidth
3. 3.33 x cores
4. 40% cost
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

3.2 TB DRAM

64 processors
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?
Exits = Stalls

No exits = no overhead \textit{[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:

\[
\begin{align*}
\text{cost}_{s1} &= w_1 f_1 i_1 + w_2 f_2 i_2 + \ldots + w_n f_n i_n \\
\text{cost}_{s2} &= w_1 f_1 i_1 + w_2 f_2 i_2 + \ldots + w_n f_n i_n \\
\text{cost}_{\text{selected}} &= \min(\text{costs}^*) \quad \text{[migrate a vcpu, or move/copy page]}
\end{align*}
\]

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 delivered 47x to 62x speedup over HDD.
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 `mysql` 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

<table>
<thead>
<tr>
<th>Experiments</th>
<th>mysql memory config</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>SSD</td>
<td>400 GB, no migration</td>
<td>1 TB, with migration</td>
<td></td>
</tr>
<tr>
<td>innodb cache</td>
<td>1 TB</td>
<td>0</td>
<td>400 GB</td>
<td>400 GB</td>
</tr>
<tr>
<td>mysql memory</td>
<td>1 TB/table</td>
<td>0</td>
<td>0</td>
<td>600 GB</td>
</tr>
</tbody>
</table>
The experiment

i=tpch

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

- run from SSD (~80GB)
- run from 400GB allocated and instantiated memory no vcpu migration
- 1 TB allocated and instantiated with vcpu migration
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.
Difficultly: 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