# Designing for performance with OpenVMS

OpenVMS Bootcamp 2017

Session 234

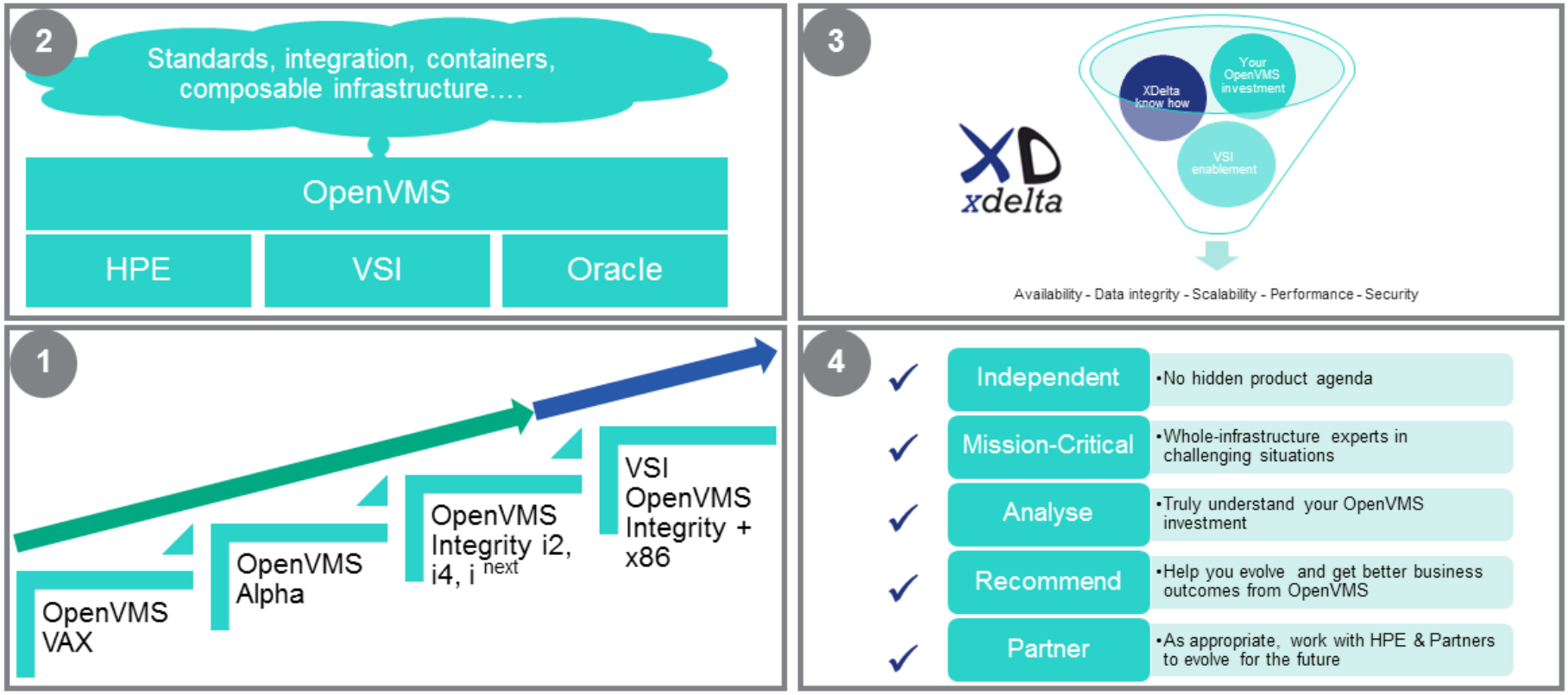
Colin Butcher CEng FBCS CITP
Technical director, XDelta Limited
www.xdelta.co.uk

# XDelta: Who we are

vms Software

- VSI Professional Services Alliance member
- Independent consulting engineers since 1996:
  - UK based with international reach
  - Delivering OpenVMS based systems for 30+ years
- Technical leadership for business-critical systems
  - Design, planning and implementation
  - Mentoring and skills transfer
  - Systems engineering background
- Gartner (2009):
  - Identified XDelta as one of few companies world-wide capable of OpenVMS platform migration projects

# XDelta - a trusted advisor to advance your critical OpenVMS application infrastructure




Hewlett Packard Enterprise

# Agenda

- Principles of performance
- Designing and implementing scalable software
- Making use of multiple CPUs (cores and hyperthreads)
- Making use of memory
- Making use of the storage IO subsystem
- Making use of the network IO subsystem
- The importance of testing

# Abstraction layers

"All problems in computing can be solved by introducing another layer of abstraction."

"Most problems in computing are caused by too many layers of complexity."

We need to strike a balance that is appropriate for the kinds of systems we're building.

# Using abstraction layers

- What looks like your dedicated resource is just a slice of a much bigger thing over which you may have little control:
  - What looks like a network isn't the whole network
  - What looks like a disc isn't a disc
  - What looks like memory isn't all of memory
  - What looks like CPUs aren't all the CPUs
- The operating system allocates and manages machine resources
- It's even more complicated in a virtualised environment

# The “Hall of mirrors”

- You can’t see everything that’s going on
- The view is often distorted
- Hiding things makes it easier to deal with the bits you’re interested in
- Hiding things makes it much harder to understand what’s happening, especially with performance related problems

# Performance characteristics

- Bandwidth – determines throughput
  - It's not just "speed", it's "units of stuff per second"
- Latency – determines response time
  - Determines how much data is in transit
  - "data in transit" is at risk if there is a failure
- "diff latency" (variation of latency with respect to time) or "jitter" - determines predictability of response
  - Important for establishing timeout values
  - Latency fluctuations will cause failures under peak load

# Capacity and scalability

- Contention and saturation – running out of capacity
  - What else are we sharing our capacity with ?
  - Queuing theory

- Increasing the capacity of the overall system:
  - “Scale up” or “vertical scaling” – adding resources to a machine or buying a bigger machine (CPU count, memory, I/O adapters, etc.)
  - “Scale out” or “horizontal scaling” - adding more machines

# Current technology trends - parallelism

- Move from low core count, high clock rate processors to high core count, low clock rate processors
- Implicit assumption is that parallelism can be achieved
- Algorithm design is key
- Don’t leave it all to the compilers
- Serialisation, synchronisation and intercommunication

# Parallelism and scalability

- Understand how the applications could break down into parallel streams of execution:
  - Some will be capable of being split into many small elements with little interaction between the parallel streams of execution
  - Some will require very high interconnectivity between the parallel streams of execution
  - Some will require high-throughput single-stream processing

- Understand scalability – do as much as possible once only, do little as possible every time

# Writing scalable code

- Check for code making assumptions that the system is a uniprocessor machine:
  - Flags controlling access to an entire global section
  - Loops polling for flag status changes (spinlocks)
  - Data structures not protected from operations that may happen in parallel instead of sequentially
- Use the lock manager to serialise and synchronise access to data structures
- Minimise wait states by having appropriate granularity of access to data structures
- Take null locks out, then simply convert them as needed

# Code paths and data flows

- How does your code scale up ?
- Separate out static data from dynamic data
- Minimise frequently executed code paths
- How to reduce impact on system and surrounding network and storage infrastructure ?

# IO performance v CPU performance

- Physical I/O operations typically takes a few milliseconds to complete

- We can execute a lot of CPU clock cycles in a few milliseconds (1 GHz = 1 nanosecond, thus 1 million clock cycles per millisecond)!

# Analysis tools and instrumentation

- Code analysis tools can help with finding interactions and heavily executed code paths
- Build instrumentation into your software, then you don't change its behaviour by adding temporary code
- Pay attention to state transitions and event timing
- Synchronisation and wait states are expensive

# Compilers

- Generate the Instruction and Data streams for processing by the system
- Different types of instructions and data are split out into separate sections (shared data, read-only data, local read-write data etc.) for use by the linker
- Generate code for a specific machine architecture
- Optimisation re-orders the code to take advantage of hardware parallelism and processing efficiencies
- Generate debug information
- Linker lays out the image address space and provides hooks for the image activator

# Limits to software performance

- Ability to make use of hardware parallelism
- Granularity of data structures
- Synchronisation techniques
- Serialisation techniques
- Scalability techniques
- Compilers
- Application design
- Designing and writing very good code requires very good programmers

# Software considerations on Integrity

- Compilers are the key to performance
- Use the documented mechanisms provided by the operating system – read the release notes and new features, then use the current mechanisms
- Exception handling (LIB$SIGNAL etc.)
- Data alignment (unexpected alignment faults)
- Floating Point format (IEEE by default)
- Implicit assumptions
- Debugging, ELF & DWARF formats
- Integrity calling standard and register usage

# Java Virtual Machines

- Java is not a compiled language running native instructions –Java run-time environment (Java Virtual Machine)
- The Java run-time environment uses significant amounts of memory and performs “garbage collection” intermittently to remove objects no longer in use
- Tuning a system to run Java well can be “interesting”

# Making use of CPU

- Hyperthreading – very workload dependent
- Fastpath IO devices and distributed interrupt handling
- Introduce parallelism into your code flow and batch jobs where possible
- Dedicated CPU for lock manager (local locking)
- Compression and encryption, eg: SCS compression, BACKUP compression
- QUANTUM, workload dependent – many SYSGEN parameter defaults changed in V8.2
- Power management

# Making use of memory

- NUMA – “mostly UMA” is a good starting point
- Use memory (XFC, resident images, DECram etc.)
- Revisit working set sizes - WSMAX and process quotas
  Use RMS global buffers
- Revisit RMS system defaults
- GH regions – map lots of memory with a small number of page table entries
- INSTALL /RESIDENT and GH region size
- 64bit P2 space and memory reservations
- DECram and HBVS to disc

# Making use of storage IO

- FC bandwidth is important – what else are you sharing your storage bandwidth with? Why?
- Rotational latency no longer matters, nor does balancing the IO load to the spindles
- Array controller cache size and volume characteristics (cluster factor, extend quantity, volume expansion etc.)
- HBVS – only shadow what you really need to
- HBVS – many shadow sets let you control how rapidly shadow copying proceeds during recovery
- HBVS mini-copy and mini-merge policies
- HBVS block count to match array controller cache size

# Making use of network IO

- Network bandwidth and latency matter:
  - Use jumbo frames
  - Split the traffic across multiple NICs
  - What else are you sharing your bandwidth with ?
  - Are the switches over-subscribed ?
  - Need to avoid retransmits
- Multiple path networks:
  - Packets may not arrive in the order in which they were sent
- Beware bus speed limitations

# Alignment faults

- Unexpected alignment faults on Integrity are expensive and affect the entire system performance
- Use the appropriate compiler switches if they are available (not all compilers have this feature)
- MONITOR ALIGN will show if you have issues
- Alignment fault tracing (with SDA)
- Look for high MP SYNC (or disguised MP SYNC which may show up as high INTERRUPT or KERNEL modes)

# Contention for resources

- Many processes running in parallel can create contention for access to data files and shared data structures
- Look for high MP SYNC (or disguised MP SYNC which may show up as high INTERRUPT or KERNEL modes)
- Fast systems with a high level of parallelism can create conditions where a lot of things ”bunch up” and the overall effect is to slow the whole system down
- Look closely at the workload and flow of activities through the system

# Exception handling

- Exception handling (LIB$SIGNAL etc.) is a more expensive mechanism on Integrity than it was on Alpha
- Consider alternatives if your code makes extensive use of exception handling as part of it’s normal flow of control
- Increase stack space when using threads

# Performance engineering

- Without good data you cannot do good performance work

- Avoid guesswork - run T4 all the time
- If needed, use T4 “expert mode” and SDA extensions

- A faster machine just waits more quickly!
- Don’t make it go faster, stop it going slower
- The fastest IO is the IO you don’t do
- The fastest code is the code you don’t execute
- The idle loop is anything but idle

# Porting applications

- Setting systems up for good performance is one of the hidden aspects of porting applications
- We need systems to easily handle the normal workload
- We need systems to be capable of absorbing unexpected spikes in workload without problems
- We don’t want to spend our time managing performance and doing tuning exercises

# What worked well ?

- Dedicated CPU for lock manager made a big difference
- Re-ordering the application workflow to reduce periods of contention made a big difference
- Going through the process of building the complete system from scratch flushed out a lot of previously hidden issues
- Involving the application developers, systems people and support people with them all working together has made a big difference

# Testing

- How can we simulate realistic scenarios ?
- Test for scale, not just functionality
- Test to find out what really happens under load and under failure conditions
- Performance failures are usually transient, so how will you capture fine-grained enough data to capture a problem ?

# “It’s slow” !

- What do they mean ?
  - Is it responding poorly ?
  - Is the responsiveness varying too much ?
  - Are batch jobs running slowly ?
  - How long are key activities taking ?
- Is the expectation unreasonable ?
- Is there anything wrong at all ?

# Performance data and trend analysis

- Without data for historical comparison, how do we know what's reasonable ?
- Without data, we're guessing
- Data needs to be synchronised in time across everything
- Don't jump to conclusions – correlation does not imply causation
- Most problems are combinations of several things

# Summary

- Most people focus on performance
- Performance and availability are inextricably linked
- Good holistic design is essential
- Performance can’t be added later
- Trouble-shooting and resolution requires good data and a thorough understanding of the whole system

# Designing for performance with OpenVMS

OpenVMS Bootcamp 2017

Session 234

Colin Butcher CEng FBCS CITP
Technical director, XDelta Limited
www.xdelta.co.uk