October 2017

Update on the Port to x86-64

Clair Grant - Director of R&D
Camiel Vanderhoeven – Software Engineer
About Me

Currently (2015 -)
• Software Engineer at VMS Software, Inc.
• X86 Architecture and C++ Expert
• Working on the x86 port

Previously
• Architect and developer of the Avanti and FreeAXP emulators, and of the Open-Source ES40 emulator
• OpenVMS experience as a contractor in government, banking, automotive, healthcare, utility, transportation, weather prediction, steel production, and nuclear industry

Personal
• Married, three kids
• Collecting old hardware (www.vaxbarn.com)
• Tinkering with Electronics and FPGAs
• Wine
Agenda

• Previous VSI Boot Camps
  – 2014: Dusted off the “Porting Play Book”
  – 2015: Described the basic plan and a few details
  – 2016: Added more plan details and described the beginnings of implementation

• Today
  – Focus on implementation progress
  – What was/is difficult?
  – Work progress and what remains
Boot Contest

• What
  – Boot OpenVMS
  – Login
  – Use DIR command to get a directory listing

• Details
  – To participate, send email to Sue Skonetski
  – Include name and contact details
  – Specify date and time of first boot
  – Guidance: Q1 2018
Projects

System
- Boot Manager
- MemoryDisk
- Dump Kernel
- Device Management
- Memory Management
- Software Interrupt Services
- Debugger

Objects & Images
- Calling Standard
- Compilers
- Linker / Librarian
- Image Activator
- Stack Unwinding
- Dynamic Binary Translator
- Conditionalized Code

Cross Build
x86 Boot Camp Sessions

- Overall Project Update - Clair Grant
- Compilers - John Reagan
- Calling Standard - Richard Boylan
- Boot Manager / Dump Kernel - Gary Newsted, Richard Bishop
- Software Interrupt Services (SWIS) - Camiel Vanderhoeven
System

Architecture-Specific Work
Boot Manager

- Select Console Mode
- Analyze Devices
- Auto-Action or Enter Command Loop
- Boot System via Memory Disk
- Primary Kernel
- Dump Kernel
- Enter Console Services

What is MemoryDisk?

- ODS-5 container file with a 3-partition disk image
- Built and maintained by OpenVMS utilities
- Contains kernel files with SYMLINKS to active system
- Shared by Primary Kernel and Dump Kernel
- Located on any accessible device, including network

Status: In use on multiple platforms.
BOOT RELATED COMMANDS:

- **BOOT** <device> <sysroot> <bootflags>: Comment in quotes, max 64 characters.
- **BOOT** DKA100: Boots with default device, system root and boot flags.
- **BOOT DKA100** 9: Boots DKA100 with default system root and boot flags.
- **BOOT DKA100** 20000: Boots DKA100 with system root 2 and boot flags 0x20000.
- **BOOT #3**: Boots the third option in the Boot Options List. See OPTIONS.
- **FLAGS** <value>: Show / Set <value> VM9 System Flags. Expressed in hexadecimal.
- **ROOT** <value>: Show / Set <value> VM9 System Root. Expressed in hexadecimal.
- **OPTIONS**: Displays the VMS Boot Options List showing the last ten unique boot commands.
- **DEVICE**: Lists VMS Boot Devices and their UEFI System equivalences.

MESSAGE RELATED COMMANDS:

- **PROGRESS**: Enables Boot Progress messages. NOPRO to disable.
- **SYSSBOOT**: Enables SYSSBOOT messages. NOSSB to disable.
- **EXECINIT**: Enables EXECINIT messages. NOEXEC to disable.
- **SYSSINIT**: Enables SYSSINIT messages. NOSYSS to disable.
- **VERBOS**: Enables Extended Boot messages. NOVERB to disable.

MODE RELATED COMMANDS:

- **DETAIL**: Enables detailed BOOMGCTR> conversation. NODETAIL to disable.
- **XLDETA**: Enables XLE debugger and sets SYSSBOOT breakpoint. NOXL to disable.
- **SYSPROMPT**: Enables SYSPROMPT conversation. NOSYSP to disable.
- **NETBOOT**: Enables NETBOOT conversation. NONET to disable.
- **DUMP**: Enables the VMS Crash Dump Kernel. NODUMP to disable.
- **DUMPDEVICE**: Sets or shows the VMS Dump Device.
- **DUMPFLAGS** <value>: Show / Set <value> VM9 Dump Kernel Boot Flags. Expressed in hexadecimal.

DIAGNOSTIC COMMANDS:

- **DEVELOPER**: Enables VSI Developer Mode. NODEVEL to disable. Function varies.
- **API**: Shows API (Interrupt Controllers) list.
- **USB**: Shows USB Device list.
- **NETWORK**: Shows NETWORK Device list.
- **APIC**: Shows APIC (Interrupt Controllers) list.
- **SMBIOS**: Show SMBIOS (System Management) data.
- **GRAPHICS**: Shows Graphics diagnostics. NOGRAPH to disable.
- **MEMCHECK**: Enables Memory Config diagnostics. NOMEM to disable.
- **DEVCHECK**: Enables Device Config diagnostics. NODEV to disable.
- **KEYMAP**: Enables Keyboard Service diagnostics.

Copyright 2017 VMS Software Inc., Bolton Massachusetts, USA
VMS Software

BOOT

DEVICES

INSTALL

SHELL

MESSAGES:  
PROGRESS  
SYSBOOT  
EXECINIT  
SYSINIT  
VERBOSE

BOOT MODES:  
DETAIL>  
XLDelta>  
XDelta>  
SYSBOOT>  
NETBOOT>

AUTOACTION:  HALT

>VMS_BOOTGR-I-REVISION: X9.0-0 Build 9 - Oct 9 2017

ENABLED: Progress messages.
ENABLED: Boot Manager interaction.

>VMS_BOOTGR-I-DEVICE. Configuring System Devices...
  + 1 Network Devices (Protocol UNDI)
  + 12 Block 10 Devices

>VMS_BOOTGR-I-DEVICE. Configuring Peripheral Devices...
Scanning PCI Bus Range: [00:04.1:07]...
Added 11 additional PCI Devices discovered by bus scan.

Configured 14 PCI/e Devices.
Assigning VMS Device Names...
Assigning VMS Controller Letters...
Assigning VMS Unit Numbers...
Assigning VMS Network Devices...
Retrieving Device Information...

BOOTGR DEVICE: DNA0 (fs0)

BOOTGR> PAGE

   ENABLED PAGE scrolling mode.

BOOTGR> B

BOOT DESTINATION DEVICE: DNA0 (fs0) VMSUSBSTICK

DEFAULT BOOT COMMAND: BOOT DNA0 0 1000034

>VMS_BOOTGR-W-MAIN. DISABLED Crash Dumps.

LOAD PATH:
  /dev/pci(0x0)/pci(0x1D,0x0)/USB(0x0,0x0)/USB(0x4,0x0)/hd(1,GPT,40985391-9DF8-11E7-B56F-9C8E9935AD96,0x10CE0,0x3E800)

>VMS_BOOTGR-I-MAIN. Allocating Kernel Memory...

ADDRESS SPACE ALLOCATION:

MAIN KERNEL SYSBOOT: PA Floor: 0x00400000, Ceiling: 0x006FFFFF, Size: 0x00200000 (2MB)
MAIN KERNEL HWRP: PA Floor: 0x00800000, Ceiling: 0x00A00000, Size: 0x00200000 (1MB), Actual: 0x00050000

MEMORYDISK: PA Floor: 0x01400000, Ceiling: 0x113FFFFFF, Size: 0x100000000 (256MB)

KERNEL BASE STRUCTURE: PA Floor: 0x00200000, Ceiling: 0x002FFFFFF

---

Copyright 2017 VMS Software Inc., Bolton Massachusetts, USA
Welcome to VSI OpenVMS

Parameter passed from the boot manager to SYSBOOT:

NAMEPB:
0x00000000.00000000.00000000.00005800

Key locations and sizes:
Kernel Base: 0x00000000.00000000.00000000.00000000 size 0x00000000.00000000.00000000.00000000
Interruptable: 0x00000000.00000000.00000000.00000000 size 0x00000000.00000000.00000000.00000000
SYSTAB: 0x00000000.00000000.00000000.00000000 size 0x00000000.00000000.00000000.00000000
Memory Disk: 0x00000000.00000000.00000000.00000000 size 0x00000000.00000000.00000000.00000000
SRPB flags address 0x00041504

Entering boot$sysboot_x86
Entering boot$init_swrb
Leaving boot$init_swrb
Entering boot$check_cpu
Leaving boot$check_cpu

MEMDISKISMOUNT, Boot memory disk mounted
X86_SYSBOOT-1-LDOAPRAM, Loading parameter file X86_64VMSSYS.PAR
Entering bfs$open_file

Parameter file is 11264 bytes long (22 blocks)
boo$loadBootfile: loading parameter file
boo$usefile: parameter file read in successfully

X86_SYSBOOT-1-LDOAPRAM, Loaded file (X86_64VMSSYS.PAR)

MEMDISKISMOUNT, Boot memory disk dismounted
Entering boot$init_memalc
Entering boot$init_memory_variables
Leaving boot$init_memory_variables
Entering boot$calc_max_pfn
Best PPFN memory ranges: 20200000 40003FFF 0 21FDFFFF
minPFN 202000, maxPFN 400033, minPFN 0, maxPFN 21FDFF, memsize 1F9804

Entering boot$calc_max_pfn
Entering bootbuild_page_tables
MAXPHYADDR is 35 bits, Max linear address is 48 bits
Entering boot$find_free_pfns
Leaving boot$find_free_pfns

Page space base addr fff8000000000000
Leaving bootbuild_page_tables

Entering bootbuild_allocation_bitmap
Leaving bootbuild_allocation_bitmap

Entering boot$find_free_pfns
Leaving boot$find_free_pfns

Entering boot$check_va
Leaving boot$check_va

X86_SYSBOOT-1-ALLOCMAP1, Allocation bitmap built
Leaving bootbuild_allocation_bitmap and boot$init_memalc
Press Enter to continue

Creating the PPFN memory map
Entering bootcreate_pfn_memory_map
Entering sort_syi_build_pfn_map
count 1F9804, phygcnt 1F9804, mem_limit FFFFFFFFFFFFF000
Leaving sort_syi_build_pfn_map
Leaving bootcreate_pfn_memory_map
Leaving bootbuild001 PPFN memory map created

Creating the SO space page tables
Entering boot$init_s0_space
Leaving boot$init_s0_space
SO space page tables created
Remapping memory disk to S2 space
Entering bootmap_memorydisk
Memory disk pa = 00000000001400000, size = 10000000 bytes

(PAGE)
Always Boot from Memory Disk – Why?

• Why did we undertake this large and complicated project?
  – Increase maintainability - one boot method regardless of source device
  – Eliminate writing of OpenVMS boot drivers
  – Eliminate modifying (or replacing) primitive file system

• Other Factors
  – Take advantage of UEFI capabilities, especially I/O
  – This opportunity may never exist again

Status: 95+% done, only final details of booting into a cluster remain
Dump Kernel

- MemoryDisk dictated the need for a new way to handle crash dump
- User-mode program with some kernel-mode routines
- It “replaces” STARTUP.COM in the standard boot sequence
- Everything the Dump Kernel needs is in the MemoryDisk
- Writes raw/compressed full/selective dumps to system disk or DOSD

**Status:** We have debugged everything we can on Itanium and will do final verification work on x86 when enough of OpenVMS is running.
$ run crash_test

Initiating crash at 29-AUG-2017 13:00:52.08...

$IDE-I-DUMBOOT, Booting the Crash Dump Kernel

***** OpenVMS T64 Operating System X600-7MV - BUCHECK *****

** Bugcheck code = 000000C4: SSR/EXCEPT, Unexpected system service exception
** Crash CPU: 00000000  Primary CPU: 00000000  Node Name: POTATO
** Highest CPU number: 00000003
** Active CPUs: 00000000,00000004
** Current Process: "SYSTEM"
** Current PSB ID: 00000000
** Image Name: $IDE-I-DUMBOOT-IDE-I-DUMBOOT-IDE-I-DUMBOOT
** Crash Time: 29-AUG-2017 13:00:52.39
**
** Dumping error logs to the system disk ($140G0A16)
** Error logs dumped to $140G0A16:[SYS0, SYSEX, SYS/DUMP.DMP]
** Dumping memory to the system disk ($140G0A16)

***** Starting compressed selective memory dump at 29-AUG-2017 13:00:58.41 *****

... Dumping...

** System space, key processes, and key gshare pages have been dumped.
** Now dumping remaining processes and global pages...

** Memory dumped to $140G0A16:[SYS0, SYSEX, SYS/DUMP.DMP]

***** Completed compressed selective memory dump at 29-AUG-2017 13:01:15.63 *****

** Time to initiate memory dump: 5.42
** Time to write memory dump: 12.61

***** Primary HALTED with code HARD-HALTS, WARM-REBOOT *****

Displays time, then $CMKRNL & ACCVIO

Starting timestamp

Ending timestamp

Statistics

New Error log details

Starting timestamp

Ending timestamp

Statistics
Memory Management

• Challenges
  – OpenVMS-style page protections for kernel, exec, super, user
  – Designing for 4-level and 5-level paging
  – 2MB and 1GB pages
  – Change to traditional paging mechanism and access

• Status
  – SYSBOOT: done (compiled and linked in x-build)
    • Get memory descriptors from the boot manager
    • Set up paging mechanisms
  – Next up:
    • Create general page management routines
    • Fix code that manages pages on their own
Device Management

• ACPI
  – Very recent download (7/28/17), plus a significant bug fix
  – In next Itanium release, but no additional use on Itanium
  – For x86, exclusively use ACPI device info for system startup configuration (no more comprehensive OpenVMS device probing)

• CPU
  – Use ACPI’s CPU numbering
  – Support up to 1024 CPUs
  – Currently researching the starting and stopping individual CPUs

• Paravirtualization
  – Enhanced I/O performance in certain virtual machines, e.g. kvm, but not VirtualBox
  – Design Spec complete, implementation in progress
Software Interrupt Services

- New Data Structures
- MTPR / MFPR
- Exceptions
- System Service Dispatching
- Interrupts
- ASTs
- Mode Switching
- Context Switching
- Performance Builds

OpenVMS expectations

Platform features
OpenVMS Assumes Things…

- VAX/VMS was designed in tandem with the VAX hardware architecture.
- Where desirable, hardware features were added to satisfy the OS’ needs.
- A lot of OS code was written to make use of these hardware features.
What are these Assumptions?

- 4 hardware privilege modes
- Each with different page protections
- And with their own stack
- 32 Interrupt Priority Levels
- 16 for Hardware Interrupts
- 16 for Software Interrupts
- Software Interrupts are triggered *immediately* when IPL falls below the associated IPL
- Asynchronous Software Trap (AST) associated with each mode, triggered *immediately* when IPL falls below ASTDEL (equally or less privileged mode)
- The hardware provides *atomic* instructions for queue operations
- The hardware provides a set of architecturally defined Internal Processor Registers (IPRs)
How does Alpha meet these Assumptions?

• Alpha is a very **clean RISC** Architecture
• But OpenVMS was definitely in the Alpha Architecture designers’ minds
• The 4 modes OpenVMS needs are part of the basic Alpha architecture
• PALcode, code supplied by firmware that has **more privileges** than even kernel mode, and which is **uninterruptible**, provides the **flexibility** to implement OS specific features
• IPLs, Software Interrupts and ASTs are implemented through a combination of hardware support and PALcode
• Atomic queue instructions are provided by PALcode
• PALcode also provides the mapping from IPRs as expected by OpenVMS to the hardware implementation’s IPRs
So how about Itanium Hardware?

• Very different story, Itanium’s design was finished before OpenVMS as an OS was considered
• Offers the 4 modes OpenVMS needs
• The TPR (Task Priority Register) provides an IPL-like mechanism for hardware interrupts only
• No compatible software interrupt mechanism or ASTs
• No atomic queue instructions
• No OpenVMS-compatible IPRs
Hence, SWIS

- SWIS (Software Interrupt Services) is a piece of low-level OS code that is involved in mode changes.
- SWIS implements the software interrupt and AST support required by OpenVMS, using hardware support as available.
- Other code in the OS (with some special support from the SWIS code to ensure atomicity) provides atomic queue instructions.
- A combination of code in SWIS and other code in the OS provides OpenVMS-compatible IPRs.
- SWIS makes the Itanium CPU look more like a VAX to the rest of the OS.
Bridge Function

SWIS bridges the gap between the assumptions made by the rest of the OS to the features supported by the hardware.
SWIS on X86-64

• Because a **similar mismatch** exists between OpenVMS’ assumptions and the hardware-provided features, SWIS will be ported to X86-64.
• **Ported** means mostly re-written here, as the **provided features are very different** between Itanium and X86-64.
• On X86-64, SWIS will have to do more, as the X86-64 architecture does **not** provide the 4 mode support OpenVMS needs.
• Because of this, SWIS on X86 will not only be active when transitioning from an inner mode to an outer mode, but **also** when transitioning from an outer mode to an inner mode.
• Also because of this, SWIS now needs to become involved in **memory management** (in a supporting role).
• There’s good news too: the Itanium architecture has some features that are very complex to manage (think RSE), that are **absent** in X86-64.
Swis on X86-64

OpenVMS Expects:
- 4 Modes, different page protections, separate stacks
- 32 IPLs (16 h/w, 16 s/w)
- Software interrupts tied to IPLs
- Per-process, per-mode ASTs, delivered when below ASTDEL
- Atomic queue instructions
- VAX-like IPRs

X86-64 Offers:
- 2 rings, different page protections, separate stacks
- 14 hardware TPR’s, mask off hardware interrupts in groups of 16
- Software interrupts unaffected by TPR’s. No IPL’s
- No AST-like concept at all
- No atomic queue instructions
- X86-64 IPRs
Design Phase

SWIS for X86-64 was designed over a period of 1.5 years (1 year part-time, 0.5 years full-time), in several phases:

- **Basic** design (not detailed enough to base implementation on)
- Detailed design for **System Service** dispatching
- Detailed design for **Hardware Interrupt and Exception** handling
- Detailed design for **Software Interrupts and ASTs**
- Detailed design for **Processes and Kernel Threads**
Design Review Phase

- **Partial** reviews as the design progressed
- In-depth **3-day** review between myself and Burns Fisher
- This one turned up a design flaw that could have enabled unprivileged code to bring down the system
- Complete **walk-through** and review in one of our weekly X86-64 engineering meetings
- A lot of the content in this presentation is based on the slides I prepared for that walk-through
Implementation Phase

Implementation started in May 2017, broken down into different parts:

• Quick and Dirty Exception Handling for early code that needs something
• Data Structure Definitions
• VAX/Alpha IPRs
• Hardware Interrupts and Exceptions
• System Services
• Software Interrupts
• ASTs
• Initialization
• Processes and Scheduling
2. **SYSTEM_PRIMITIVES execlet builds**

- Compatibility build, works on any x86-64 CPU we support
- Performance builds, optimized for CPUs that have support for one or more of the following:
  1. Address Space Numbers (PCIDs) in TLB
  2. RDGSBASE instruction
  3. XSAVES/XRSTORS instructions for saving/restoring extended ("floating point") registers (MMX, SSE, AVX)
- Highest Performance build targets Intel processors made after 2013 (Ivy Bridge and beyond).
SWIS Data Structure

- One per CPU, stays with CPU over the lifetime of the system
- Only CPU-specific data structure that can be found directly
- Has a different virtual address for each CPU
- Pointed to by GS segment register
Mode “Components”

- Processor ring (0 for K, 3 for ESU)
- Stack pointer
- Address Space Number
- Page Table Base
- Current mode as recorded in the SWIS data structure

- A mode is “canonical” when all the above are in agreement
- SWIS should be the only code that ever sees non-canonical modes

- We prototyped this on Itanium
Basics of Mode Switching

• Interrupt or SYSCALL instruction
  1. Switches CS and SS to ring 0
  2. Switches to the kernel-mode stack (interrupt only, not SYSCALL)
  3. Disables interrupts
• Get fully into kernel mode (ASN, PTBR, stack, DS, ES)
• Going in? -> Build return frame on stack
• Going out? -> Deliver SwInts and ASTs as needed
• Get into destination mode (ASN, PTBR, stack, DS, ES)
• IRET or SYSRET instruction
  1. Switches CS and SS to ring 3
  2. Switches to the outer-mode stack (IRET only, not SYSRET)
  3. Enables interrupts
XDELTA-lite (XLDELTA) Debugger

• Wanted something, however primitive, as early as possible
  - Started from scratch, written in C and a little assembler
  - Follows XDELTA syntax
  - Linked into SYSBOOT

• Current Capabilities
  - Set and proceed from breakpoints
  - Examine and deposit register
  - Examine and deposit memory location
  - Examine multiple instructions
  - Examine instruction and set breakpoint
  - List breakpoints

• XDELTA vs. XLDELTA?

Status: In use, may add additional capabilities
Virtual Machine Guest

Current priority order

1. VirtualBox
2. kvm
3. Hyper-V
4. xen
5. VMware

FAQ: Will OpenVMS be a hypervisor?
   A: Extremely unlikely
Objects & Images

Image Building and Execution
Calling Standard

• Started with AMD-64 runtime conventions
• Deviated only where absolutely necessary
• Problem #1
  – Standard assumes all within-the-image addressing can be done PC-relative
  – OpenVMS Image Activator may change relative distances between image sections
  – Solution: Attach a linkage table to each code segment and address all data through it
• Problem #2
  – Need to preserve 32b addressability when procedures are in P2 or S2
  – Solution: Create 32b-addressable stubs that forward calls to the procedures
• Status
  – Satisfies all current development needs
  – Remaining work: address unwinding, debugger, and translated code issues as they arise
Inner Workings of GEM-based Compilers

- **Frontend**
  - Parses command line
  - Validates source code
  - Generates GEM intermediate representation (IR)

- **GEM**
  - Interprets IR
  - Generates target architecture object file
x86_64 OpenVMS Image Building

- C
- BLISS
- FORTRAN
- BASIC
- COBOL
- PASCAL
- XMACRO
- C++
- Ada

Internal Representation Converter
Backend Code Generator

GEM IR → G2L → LLVM IR → LLVM → .OBJ* → LIBRARIAN → .OLB → LINKER → .EXE*

* = ELF like Itanium
History of the Porting Play Book

Chapter 1 – Executable Images
Chapter 2 – Architecture-Specific Needs (a.k.a. “The 5%”)
Chapter 3 – Compiling and Linking Everything Else (a.k.a. “The 95%”)

Started with the VAX-to-Alpha Working Design Document (WDD) TOC
Concepts and high-level approach are mostly unchanged
Alpha-to-Itanium - ELF / DWARF
Itanium-to-X86 - LLVM
LINKER / LIBRARIAN

• Build the IA64 and x86 linkers from the same sources
• Problem #1
  – Much of the existing code assumed a Procedure Value points to a Function Descriptor
  – On x86 a Procedure Value points to code
• Problem #2
  – Existing code did not cleanly separate architecture-specific and architecture-independent code
  – Use conditional compilation to build IA64 and x86 version
• Status
  – Links SYSBOOT, XLDELTAL, and various other test programs
  – Remaining: reorg image sections; unwind, debugger, and translated code support
• Status – Librarian: done
Image Activator

- Most of the Itanium implementation “just works”
- Differences are due to calling standard details
- Code has been written and awaits testing on x86
- New code is shared between these two components
Stack Unwinding

- Changes largely reflect Calling Standard differences, most obvious being register sets
- OpenVMS IA64 implementation is based on HP-UX IA64 unwind code
- For x86, open source portable unwind library (http://www.nongnu.org/libunwind)
- **Goal:** Preserve the current API – LIB$xxx interface
- **Implementation**
  - Embed the unwind library context in the OpenVMS invocation context block (ICB)
  - Exception processing also has an area in the ICB
  - Add OpenVMS specifics
  - Linker puts unwind info in image; image activator extracts info and creates master unwind table (MUT); unwind code references MUT
- **Status:**
  - Design complete and reviewed
  - Readying code for checkin
Alpha-to-x86 Dynamic Binary Translator

• Directly execute an Alpha image on x86
• No restrictions in terms of compiler version or operating system version of source
• Does not support privileged code translation
• **Status: working prototype on x86 linux**
  – Using selected pieces of simh as a basis for emulation
  – Running simple Alpha images on x86 linux
  – Temporary code to emulate
    • OpenVMS loader and image activator
    • some OpenVMS library routines
  – BASIC, C, COBOL, FORTRAN, and PASCAL images have been translated
  – With no optimization work, performance is about equal to an Alpha ES47
Dynamic Binary Translator Flow

- First execution
  - Runs in emulation mode
  - Creates topology file
  - Quite slow
- Each subsequent execution
  - Reads topology file
  - Generates LLVM IR
  - Runs natively
  - Updates topology file, if needed

Dhrystone: microseconds/run

- Native 0.2
- Emulated 14.1
- Translated 0.2

Next Steps

- Synchronize topology updates (multiple users)
- Security of topology file
- Image activator integration
- Improve performance
- Translate a VESTed image – looks to be difficult
Conditionalized Code

- 776 modules needed for First Boot
  - 672 are done
  - The rest will happen as architecture-specific projects complete
- An additional 2500 are needed for the full system build
- Plus, DECnet, DECwindows, and Layered Products
Cross Build

• **Build on Itanium, target x86**
  – Builds done roughly weekly
  – Let the build tell us what we do not already know
  – Building everything
  – At some point will ignore components not needed for First Boot

• **Tools in place**
  – BLISS, C, XMACRO, assembler
  – Linker, Librarian, Analyze, SDL

• **Status**
  – Concentrating on INIT through ASSEM phases
  – Reducing “noise” with each iteration
FAQ: What are the visible differences that will come with x86-64 OpenVMS?

• Applications: none that we know of now

• Interactive users and command procedures: none that we know of now

• System managers: new utility to update the MemoryDisk
Thank You

To learn more please contact us:
vmssoftware.com
info@vmssoftware.com
+1.978.451.0110