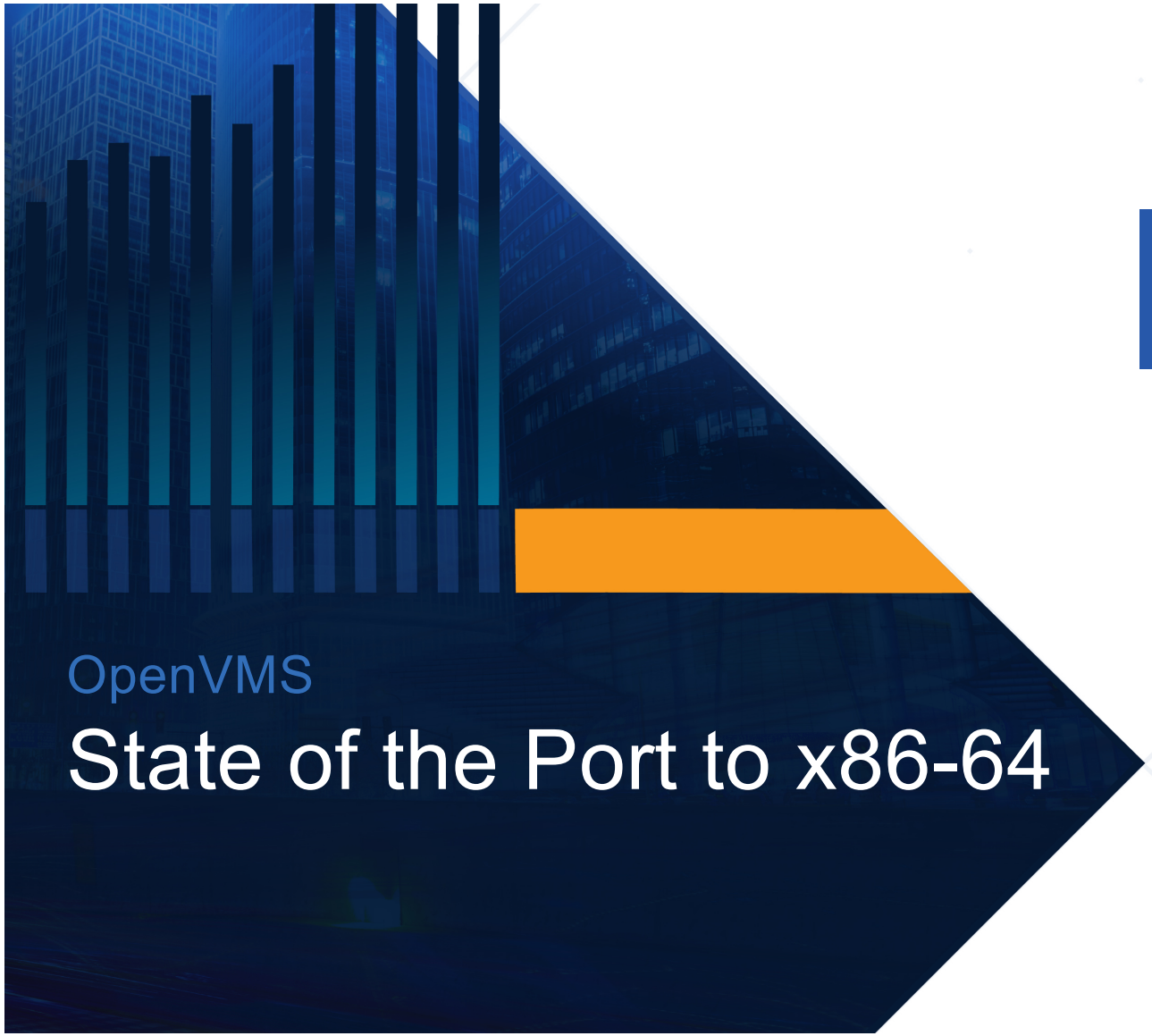


vms Software

OpenVMS

State of the Port to x86-64

October 6, 2017

Executive Summary - Development

The Development Plan consists of five strategic work areas for porting the OpenVMS operating system to the x86-64 architecture.

- OpenVMS supports nine programming languages, six of which use a DEC-developed proprietary backend code generator on both Alpha and Itanium. A converter is being created to internally connect these compiler frontends to the open source **LLVM backend code generator** which targets x86-64 as well as many other architectures. LLVM implements the most current compiler technology and associated development tools and it provides a direct path for porting to other architectures in the future. The other three compilers have their own individual pathways to the new architecture.
- Operating system components are being modified to conform to the **industry standard AMD64 ABI calling conventions**. OpenVMS moved away from proprietary conventions and formats in porting from Alpha to Itanium so there is much less work in these areas in porting to x86-64.
- As in any port to a new architecture, a number of **architecture-defined interfaces** that are critical to the inner workings of the operating system are being implemented.
- OpenVMS is currently built for Alpha and Itanium from **common source code** modules. Support for x86-64 is being added, so all conditional assembly directives must be verified.
- The boot manager for x86-64 has been upgraded to take advantage of new features and methods which did not exist when our previous implementation was created for Itanium. This will **streamline the entire boot path** and make it more flexible and maintainable.

Executive Summary - Release

The Release Plan includes multiple stages.

- Once First Boot is achieved work will progress towards an Early Adopter Kit. First Boot is defined as bringing the system up to the point of logging in and successfully executing a DIRECTORY command. That may not sound like much but most of the architecture-specific aspects of the operating system must be working as well as being able to compile, link, load, and execute privileged and non-privileged code.
- The **V9.0 Early Adopter Kit (EAK)** will be for selected ISVs and partners. There will be some system components, most layered products, and analysis tools not in place or not yet complete but it will be enough to start porting applications x86.
- The **V9.1 General EAK** will be for all ISVs, partners, and customers. It will be close to the complete system but not production quality in terms of stability or performance.
- The **V9.2 Production Release** will be the complete system as it ships today on Alpha and Itanium.

Announcing the x86 Boot Contest

- Guidance: Calendar Q1 2018
- Entry = date and time
- Prizes for the five closest
- Entries close November 30, 2017
- Entry process will be publicized soon

- **Definition of First Boot**
  - Bring up the system
  - Login
  - Execute successful DIRECTORY command

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

System Architecture

System Architecture-Specific Work

vms

Boot Manager

- Modern UEFI “console” application to construct the boot environment
- Selects Console Mode (in specified order)
  1. Full Graphical Display with Keyboard and Mouse, Touchpad or Touchscreen Input
  2. Rendered Text Display with Keyboard Input
  3. Serial IO, optionally in addition to one of the above.
- Analyzes Devices
  - ACPI-guided probe of local device scope (anything visible to UEFI)
  - Enumeration of bootable devices using OpenVMS device names (i.e. BOOT DKA100)
  - Supports 1024 Processing Units
  - Deep identification of PCI/e and USB devices
- Auto-Action or Interactive Command Mode
  - Automatic Boot for unattended operation
  - Command Mode with ~60 commands (some earmarked for developer use)

vms

Booting from MemoryDisk

- Versatile Booting via MemoryDisk
  - Boot Manager is no longer tied to the device being booted
  - Uses UEFI Block I/O Drivers instead of OpenVMS Boot Drivers
  - Can initiate a boot from any source that can serve the MemoryDisk file
  - Supports multiple forms of remote boot (Web-Server, Wireless, iSCSI, SAN, FCoE, more)
  - Substantial integrity and security checks throughout boot process
- Load Dual Kernels (Primary & Dump)
  - Boot Manager recycles “Boot Services” memory for the Dump Kernel
  - Bugcheck initiates fast-boot of Dump Kernel
  - Dump Kernel leverages “runtime” drivers and environment
- Provide Additional “Runtime” Services
  - Console IO Services for Operator and Debugger use
  - Dump Kernel Boot Requests
  - “Update” Boots
  - Graphical Services (example: Visual Register Decode)

What is the MemoryDisk?

- MemoryDisk in the eyes of the Boot Manager
  - "Mini-Kernel" ODS5 Container File (SYS$MD.DSK), pre-packaged and maintained by the OS
  - Contains everything needed to achieve viable OS boot or initial installation
  - Contains 3-partition disk image: VMS_LOW, UEFI (aka SYS$EFI.SYS), VMS_HIGH
  - Uses Symlinks during OS operation
  - Platform Firmware sees only the UEFI Partition: VMS_BOOTMGR, SYS$MD.DSK
  - Optional UEFI SHELL for systems requiring one
  - Boot Manager downloads SYS$MD.DSK into memory using UEFI Block I/O
  - Boot Manager extracts SYSBOOT.EXE using LBN Map in UEFI Partition's Boot Block
  - Boot Manager supports "local" SYSBOOT.EXE and SYS$MD.DSK for development use.
- To the Boot Manager, SYS$MD.DSK is just a file
  - Carve up memory, download the file, put it in memory, jump to it, then remain available as runtime services.
  - Carve out a second Mini-Kernel using recycled memory for the Dump Kernel. MemoryDisk is shared, only kernel structures and SYSBOOT are replicated.

Always Boot from MemoryDisk – 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 kernel-mode routines to map its virtual addresses to primary kernel memory
- Runs in an environment equivalent to STARTUP.COM
- Everything the Dump Kernel needs is in the MemoryDisk
- Writes raw/compressed full/selective dumps to system disk or DOSD
- A page of data is accessible by both primary and dump kernels
  - BUGCHECK in the primary kernel writes key data (PTBRs, BUGCHECK code, base image symbols, exception frame)
  - Dump Kernel finds everything it needs from there
- Dump Kernel reads
  - symbols from SYS$BASE_IMAGE and figures out their virtual addresses
  - memory map so it knows the valid PFNs
- **Status**: We have debugged everything we can on Itanium and will do final verification work on x86 when enough of OpenVMS is running.

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
    - Update code that manages pages on its own to use common routines

Device Management

- **ACPI**
  - Very recent download (7/28/17), plus a significant bug fix
  - New code is 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

OpenVMS Makes Many Assumptions

- VAX/VMS was designed in tandem with the VAX hardware architecture
- Code was written to take advantage of the VAX hardware features
- 4 hardware privilege modes, each with different page protections and stack
- 32 Interrupt Priority Levels, 16 for HW interrupts and 16 for SW 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)

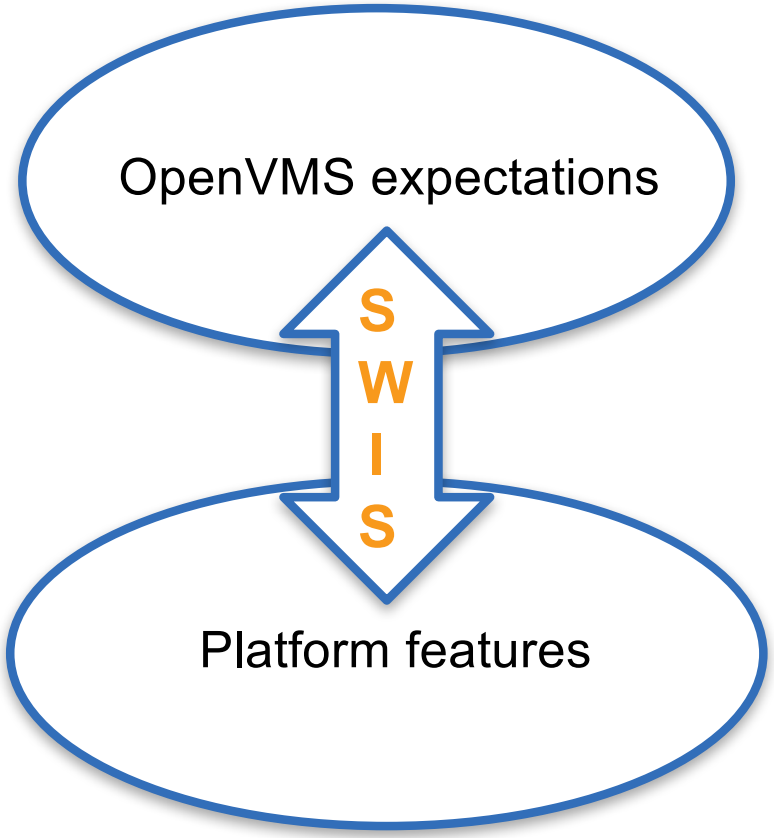
Software Interrupt Services (SWIS)

- SWIS is low-level operating system code involved in mode changes
- X86-64 architecture does not provide the 4-mode support OpenVMS assumes
- Hence, SWIS on X86-64 is active when transitioning from an inner mode to an outer mode, and when transitioning from an outer mode to an inner mode
- Also due to this, SWIS has a supporting role in memory management
- SWIS implements the software interrupt and AST support required by OpenVMS, using hardware support as available
- Other operating system code (with some special support from SWIS to ensure atomicity) provides atomic queue instructions
- A combination of SWIS code and other operating system code provides VMS-compatible IPRs
- SWIS makes the CPU look much like a VAX to the rest of the system

vms

Software Interrupt Services

- **Status**: Code is 90% written and early debugging will start within a couple weeks.



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
- **Status**: In use, may add additional capabilities

OpenVMS as a Virtual Machine Guest

The current priority order follows. Regular testing now occurs on VirtualBox and kvm.

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

FAQ: Will OpenVMS ever be a hypervisor?

ANS: Extremely unlikely

Objects & Images

Image Building & Execution

vms

Calling Standard

- An Application Binary Interface (ABI)
  - Defines what a system looks like to an assembly-language programmer
  - Is different for each CPU architecture
- OpenVMS x86 Calling Standard is based on the industry standard AMD64 ABI (used on UNIX/Linux) systems
  - Deviates from industry standard only where absolutely necessary
  - Adds a few upward compatible extensions, for example argument count
- Goals
  - Compile-and-go compatibility with IA64 OpenVMS
  - Follow industry-standard calling conventions as long as they don't conflict with the first goal
  - Don't introduce security holes into the runtime environment
  - Maximize performance where it doesn't conflict with the previous goals

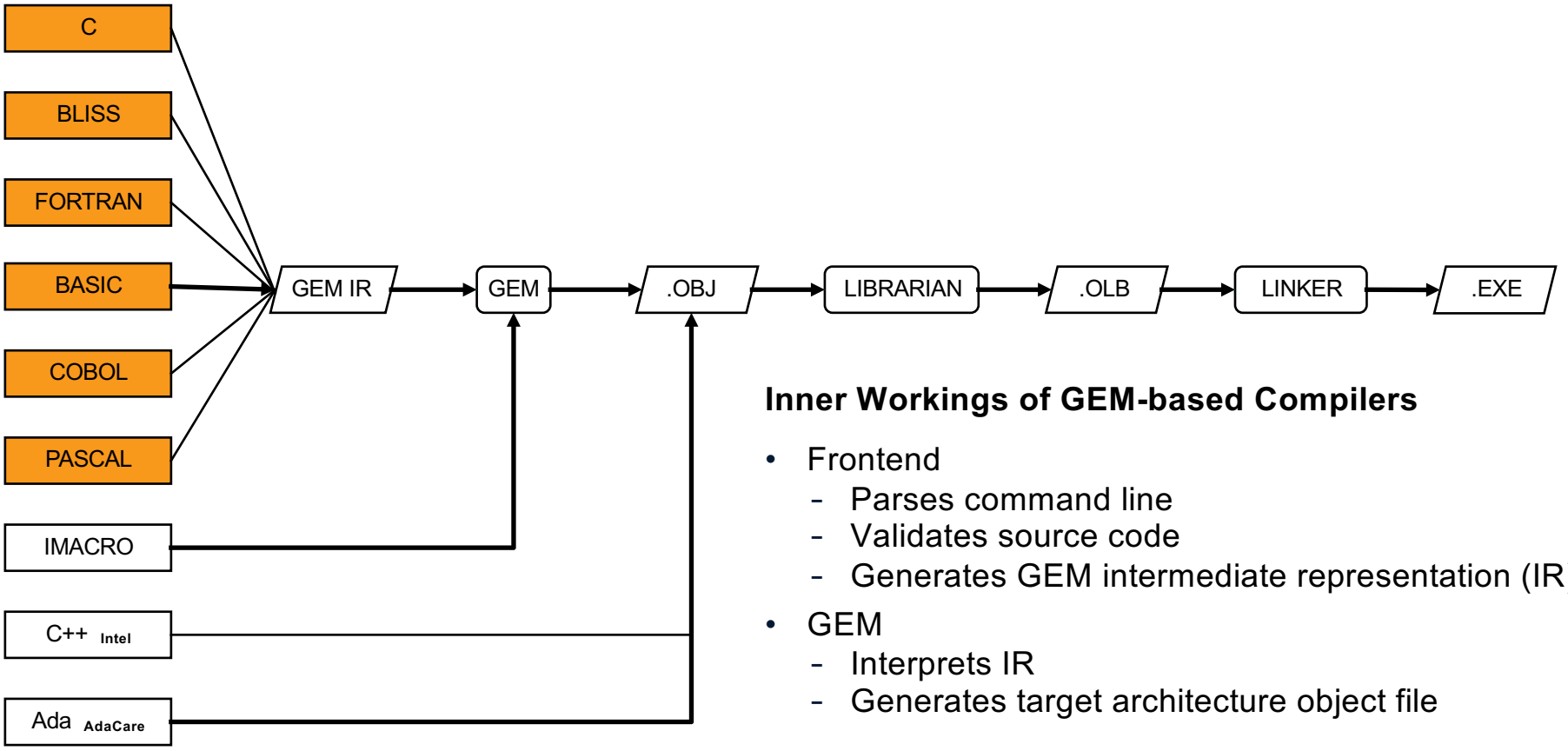
Calling Standard

- 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
  - Likely remaining work: address unwinding, debugger, and translated code issues as they arise

Itanium Compilers

- Most are GEM-based
  - Common code for command qualifiers, reading source files, writing object files, etc.
  - Generate GEM intermediate language and symbol table
  - GEM optimizes, generates code, and writes out object/listing files
  - Frontends have small knowledge of target environment
  - Frontends have common code with Alpha
- IMACRO
  - Uses GEM for command qualifiers, reading source files, writing object files, and more
  - Interfaces to GEM at a lower level meant for assemblers
  - 2/3rd common code for Alpha and Itanium
  - 1/3rd Itanium architecture specific code
- C++ based on Intel C++ compiler extended for OpenVMS

Itanium OpenVMS Image Building



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

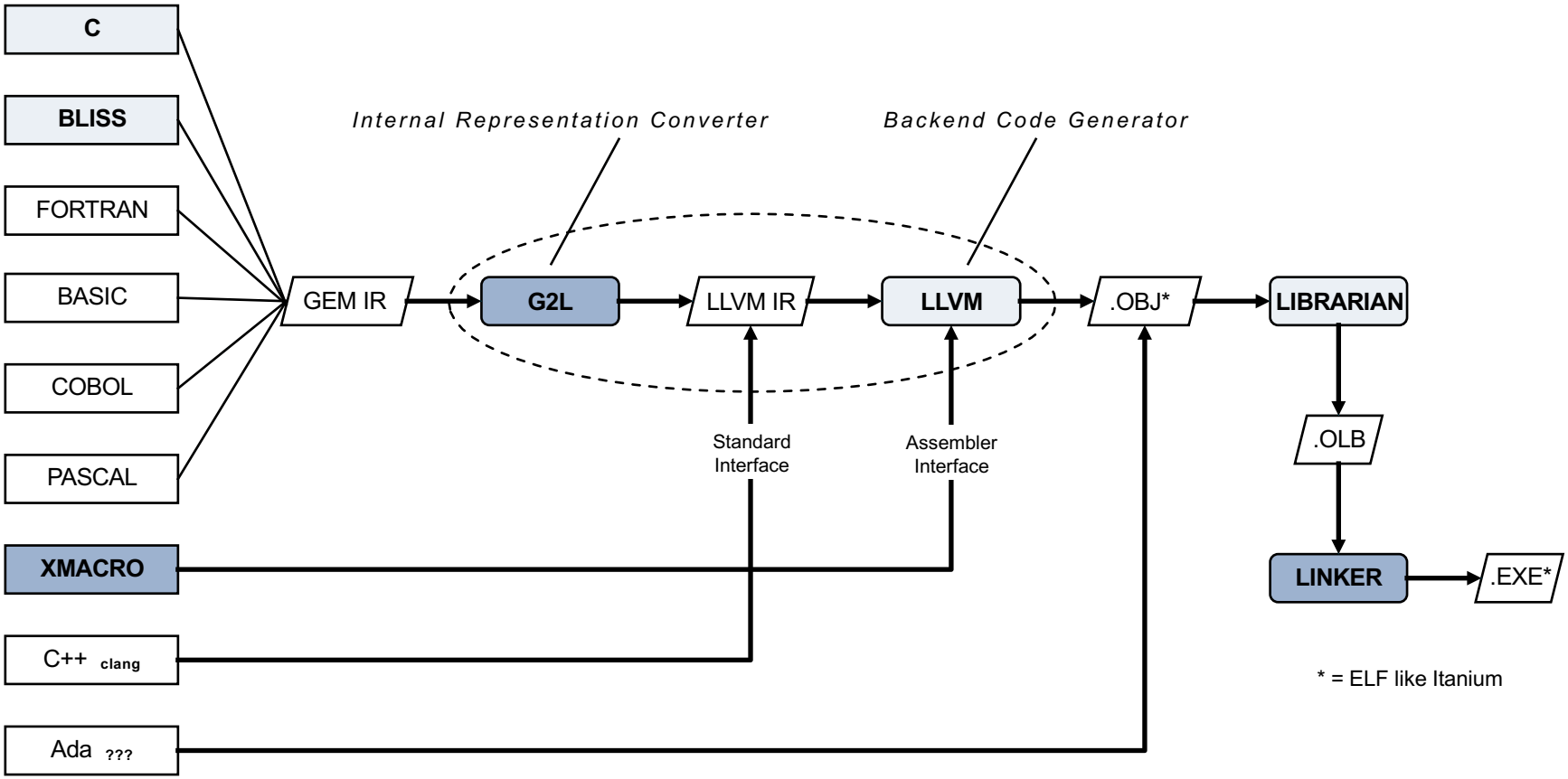
Leveraging GEM

- Not all of GEM is target-dependent
- GEM interfaces for command line and source file control are all still valid and will continue to be used
- GEM interfaces for building symbol-table, CIL and debug info will still be used to enable common code frontends
- Some GEM interfaces might need implementation changes for things like machine code listings and debug information

Leveraging LLVM

- Keeps VSI out of the “backend business” of chasing every chip release
- Large user community
- Friendly license structure
- Good documentation
- Active contributions from large companies like Apple and Google
- Active contributions from research community
- Creating x86 images
  - Major work has been in XMACRO, G2L and Linker
  - Some work required in BLISS and C frontends, LLVM and Librarian

x86_64 OpenVMS Image Building



BLISS Compiler

- Essentially unchanged from Itanium
- Almost all GEM IR implemented
- Argument list processing and non-standard LINKAGE support still in progress
- Performed some limited testing with “compile on OpenVMS” and “link and execute on Linux or MacOS”

C Compiler

- Mostly unchanged from Itanium
- Almost all GEM IR implemented
- Performed extensive testing with “compile on OpenVMS” and “link and execute on Linux”
- <stdarg.h> support for VA_LIST, etc. needed frontend changes
- Frontend has some ABI specific knowledge on argument passing
- <vararg.h> will be unsupported
- Some additional information needed by G2L converter needs new fields in a few GEM IR and symbol table nodes to aid in argument list support
- Assorted new x86-64 builtins as needed

XMACRO Compiler

- XMACRO
- Continue using GEM for command qualifiers and reading source files
- New code to generate x86-64 instructions
- Uses LLVM's assembler interface
- VAX/Alpha registers mapped to memory locations
- OS will understand how to do unwinding with these memory locations
- Interoperates with BLISS and C for non-standard linkages to minimize code changes in the OS
- Leverage x86-64 condition codes for VAX condition codes
- Will add some x86-64 specific builtins as needed

C++/clang

- Will use clang to provide up-to-date C++ standard support
- Will add OpenVMS-isms to clang much like was done did for the Intel-based C++ on Itanium such as dual-sized pointers.
- Extend the clang “driver” to use GEM for command line support
- Will use the LLVM standard C++ library (libc++) with OpenVMS additions as needed
- The CRTL headers and CRTL will need extending as well

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 versions
- **Status**
  - Links SYSBOOT, XLDELTA, 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” since the internal format of an image is ELF on both Itanium and x86
- The few differences are due to calling standard details
- Some of the new code is shared among multiple system components
- **Status**: Code has been written and awaits testing

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, as in $ RUN APP.EXE
- 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 the 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
  - Even 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

- Switch to emulation and then back to translation
- Synchronize topology updates (multiple users)
- Security of topology file
- Image activator integration
- Improve performance
- Translate a VESTed image? – looks to be doable but difficult

Cross Build

vms

Conditionalized Code

- Every source file with conditionalized code must be examined in order to guarantee that the correct code is generated for Alpha, Itanium and now x86
- 776 modules need to be verified for First Boot
  - 672 are done
  - The rest will be done as architecture-specific projects complete
- An additional 2500 files with conditionals are needed for the base system build
- Plus, DECnet, DECwindows, Compilers 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
  - For now, building everything
  - At some point we will temporarily 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

Thank You

To learn more please contact us:

vmssoftware.com
rnd@vmssoftware.com
+1.978.451.0110

vms