vms Software

Porting OpenVMS to x86-64
Update

Clair Grant
Camiel Vanderhoeven

April 8, 2016

vms Software

Porting OpenVMS to x86-64
Update

This information contains forward looking statements and is provided solely for your convenience. While the information herein is based on our current best estimates, such information is subject to change without notice.

Porting Play Book (The Plan)

Chapter 1 – Executable Images

- **Definition**: Register Mapping, Calling Standard extensions
- **Creation:** Compilers, Assembler
- **Action:** LIBRARIAN, LINKER, INSTALL, Image Activator
- **Analysis:** SDA, DEBUG/XDELTA, ANALYZE IMAGE, ANALYZE OBJECT

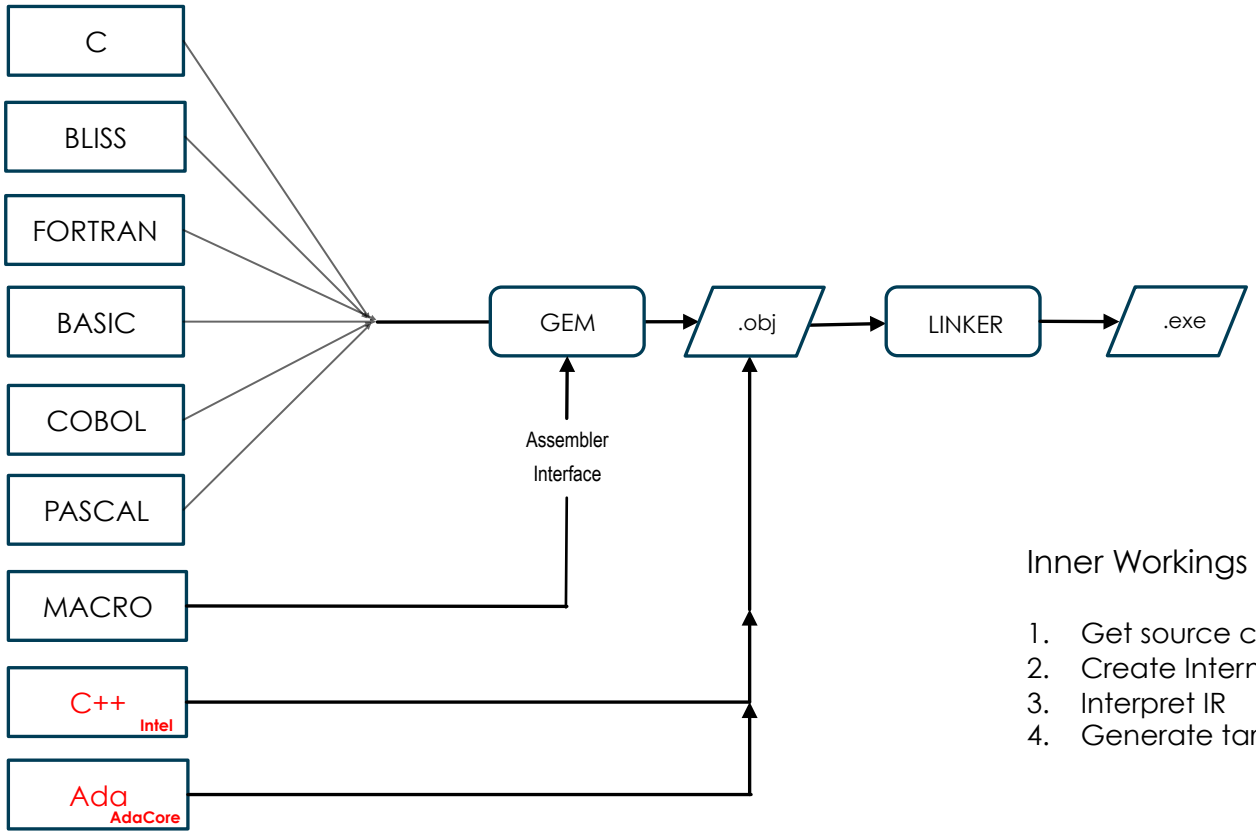
Chapter 2 – Architecture-Specific Needs (a.k.a. “The 5%”)

- Booting
- Interrupts, Exceptions
- Memory Management: protection types, access modes, address space, etc.
- Atomic Instructions
- Floating Point
- Special needs for code in assembler (e.g. VAX QUEUE instruction emulation)

Chapter 3 – Compiling and Linking Everything Else (a.k.a. “The 95%”)

- Large task but mostly mechanical
- Flush out any remaining ‘inter-routine linkage’ problems

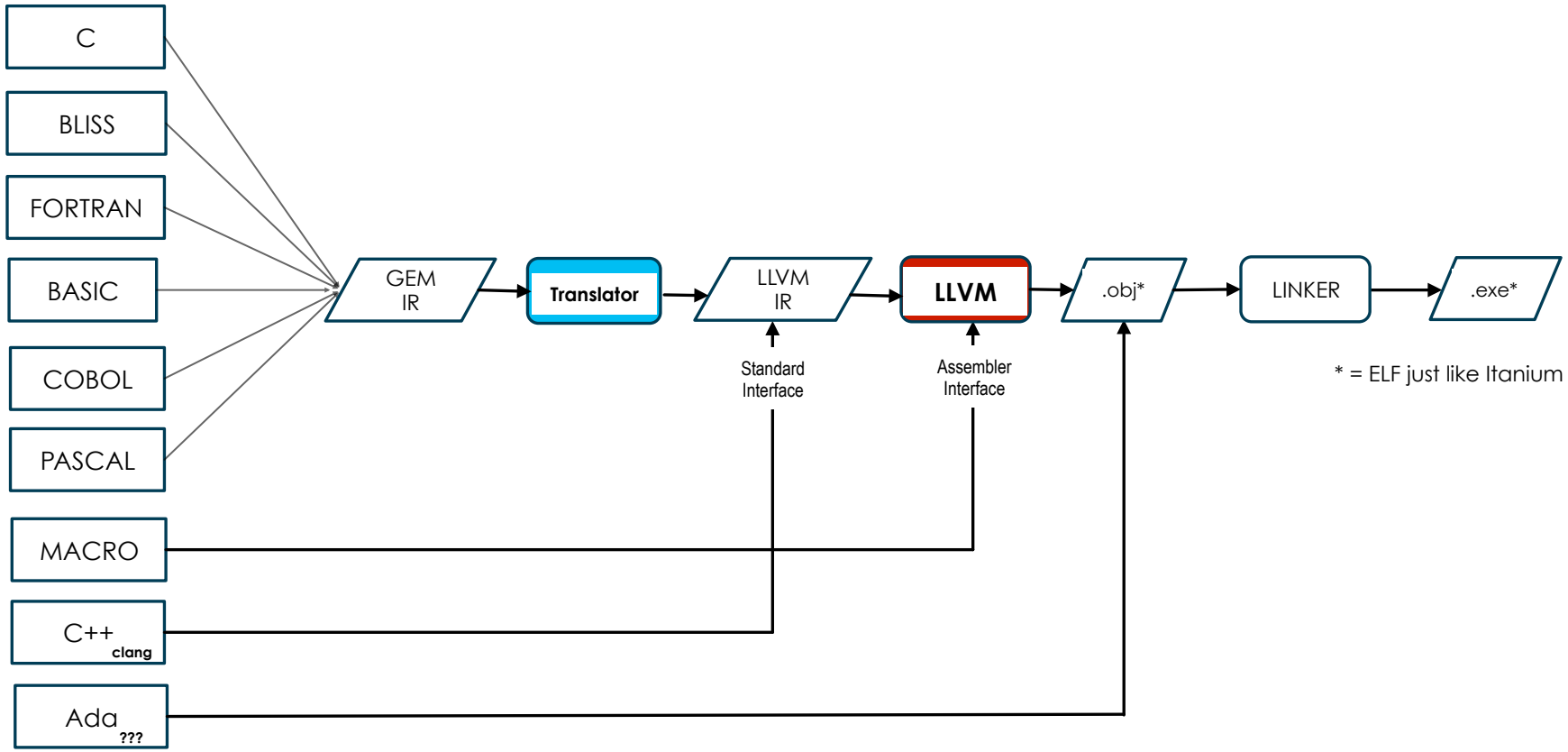
VMS Itanium Compilers and Image Building



Inner Workings of GEM

1. Get source code and command line directives
2. Create Intermediate representation (IR)
3. Interpret IR
4. Generate target object file

Future VMS Compiler Strategy



- Continue with current GEM-based frontends
- Use open source LLVM for backend code generation
- *Create internal representation (IR) translator*
- LLVM targets x86, ARM, PowerPC, MIPS, SPARC, and more

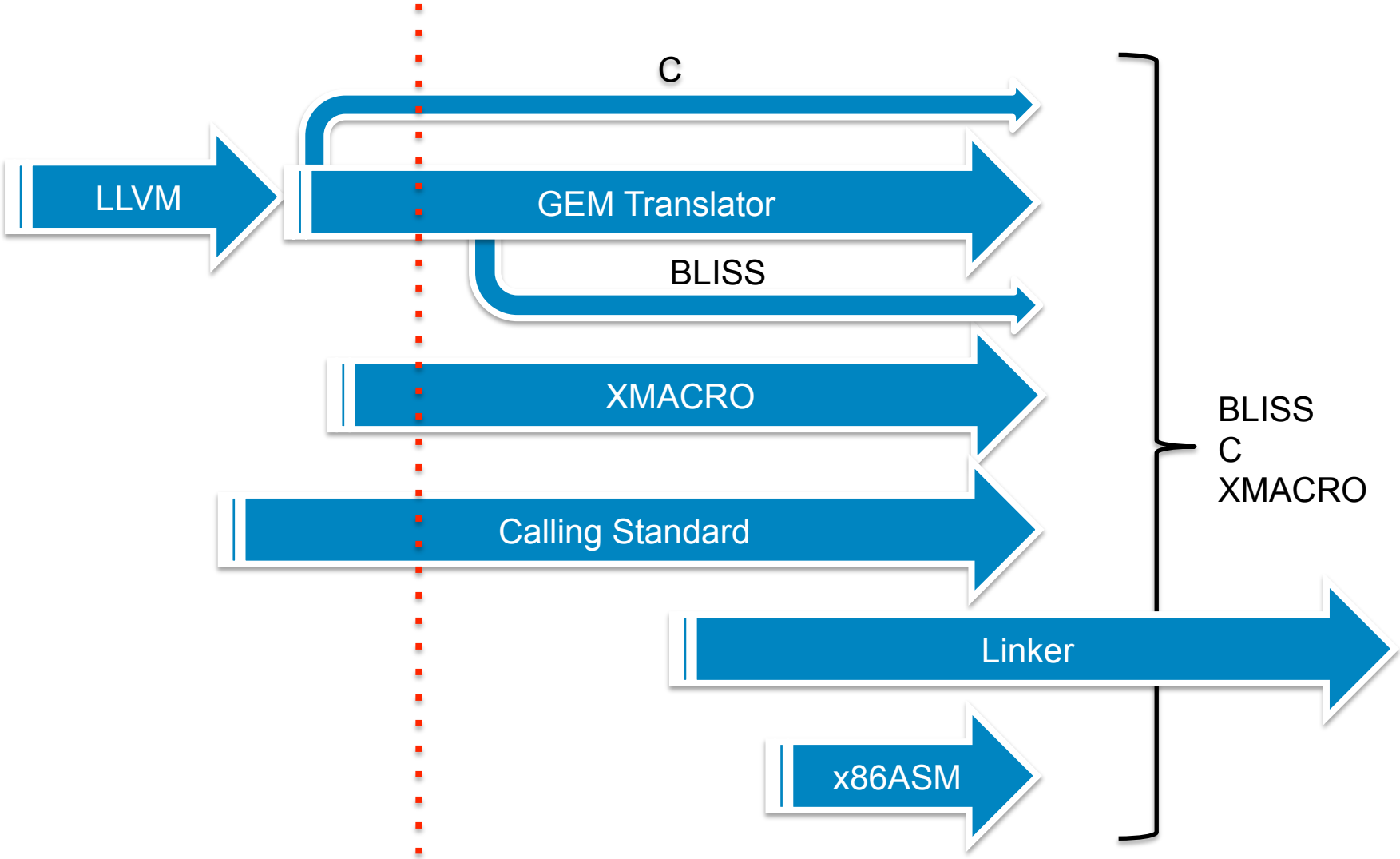
Executable Images

- Compilers
  - LLVM
    - Compiled on VMS Itanium ***COMPLETE!***
    - GEM IR to LLVM IR translator *UNDERWAY*
      - Compile with DEC C/LLVM on OpenVMS Itanium, link and run on Linux
  - XMACRO
    - Register mapping ***COMPLETE!***
    - Connect XMACRO to LLVM assembler interface *UNDERWAY*
    - Developing VAX to x86 instruction mapping *UNDERWAY*
    - Developing VAX to x86 addressing mapping *UNDERWAY*
- Calling Standard *UNDERWAY*
  - Based on AMD64 Application Binary Interface
  - Like Itanium - ELF, DWARF, unwind tables

Executable Images (continued)

- LINKER
  - Initial outline of work ***COMPLETED!***
  - Relocations and fixups – minimal work
  - Calculate virtual address space – some work
  - Pass 2 – most of the change is here
  - Writing executable/shareable images and symbol table files – small change
- LIBRARIAN - minimal work
- INSTALL, Loader, Image Activator
  - Relocations/fixups – small change
  - Share most crucial pieces of code
- Static/dynamic translator

“First Boot” (with Cross Tools) Images



Analysis Tools: Laying the Foundation

- x86 Instruction Set Decoder ***COMPLETE!***
  - 640 opcodes in total
  - Test ‘byte streams’ created for each instruction; developed/verified on linux
  - Used by SDA, DELTA/XDELTA, DEBUG, SCD, ANALYZE/OBJECT

- Evaluate LIB$IPF_CALLING_STANDARD routines; used by “stack walkers” and others
  - Invocation Context (current, previous)
  - Registers
  - Unwind data

Architecture-Specific Needs

- Boot Path
- Memory Management
- Use of Assembler

Boot Path

- VMS_LOADER.EFI
  - Using UEFI 2.3 toolkit – Itanium and x86 ***COMPLETE!***
  - Eliminate VMS boot drivers, use UEFI device drivers and memory disk ***COMPLETE!***
  - New crash dump strategy and implementation *TESTING*
  - Eliminate HPE-specific firmware interfaces ***COMPLETE!***
  - Create
    - memory descriptors ***COMPLETE!***
    - Interrupt Vector Table (IVT) *PROTOTYPING*
    - CPU enumeration list *PROTOTYPING*
    - x86-specific structures *PROTOTYPING*
  - Make better use of ACPI interface and tables *PROTOTYPING*
  - Create a graphical boot manager *PROTOTYPING*
  - X86 Machine check handling *PROTOTYPING*
- *IPB & SYSBOOT* *evaluating*

Memory Management

- Initial investigation ***COMPLETE!***
- Factoids:
  - Four levels of page tables *PROTOTYPING*
  - VMS will run in two hardware modes *PROTOTYPING*
  - Page protection requires page tables per mode
  - No PROBE instruction; look it up in page tables ***COMPLETE!***
  - Page sizes – 4KB, 2MB, 1GB

Running in Two Processor Modes

- x86 has four modes (rings) 0, 1, 2, 3.
- They do not provide the strict hierarchy of memory access protection expected by VMS.
- Example: No way to allow kernel write and prevent exec write.

- VMS will run in two hardware processor modes – privileged (0) and unprivileged (3).
- Supervisor (1) and Exec (2) memory protections will be implemented in software.
- Currently prototyping and testing two-mode operation on Itanium.

SWIS & Friends

- Many related architecture-specific details in one place: Software Interrupt Services (SWIS), Exception, AST Delivery…
- Hides the details from the rest of VMS
  - Many aspects of the Calling Standard
  - Entering a more privileged mode
  - Interrupt handling
    - Software Interrupts
    - ASTs
    - External Interrupts
  - Saved state
  - Exception frames
  - Context switching
  - System service calling

SWIS & Friends

- Conceptually architecture independent – the code is specific for each platform but it performs the same logical functions
- The largest single piece of concentrated assembler code apart from IMATHRTL
- Design finished, currently under review

Use of Assembler

- Evaluate Itanium assembler code ***COMPLETE!***
  1. Eliminate what is not needed
  2. Replace with C equivalents if possible
  3. Convert to x86 assembler
- Priority: Follow the boot path
- If better performance is needed then use assembler – difficult to predict, just let it happen and react

Virtual Machines

- Development/test environments:
  - KVM / CentOS
  - XEN
  - VirtualBox
  - VMware ESXi 6.0
  - Vmware Workstation 12
  - VMware Fusion
- Used so far for
  - Debugging VMS_LOADER.EFI
  - Prototyping and experimenting
- Paravirtualized storage drivers *implementing*
  - Starting point: HPVM driver
- Paravirtualized network drivers

Various

- VAX/Alpha/IA64 conditionalized code — *UNDERWAY*
- Non-standard calling sequences — *INVESTIGATING*
- Evaluated Ada code — ***COMPLETE!***
  - Plan to rewrite ACME in C
  - Plan to rewrite Security_Server in C++

For more information, please contact us at:

RnD@vmssoftware.com

VMS Software, Inc. • 580 Main Street • Bolton MA 01740 • +1 978 451 0110