State of the Port to x86_64 April 2017

April 3 , 2017

Update Topics

- Executive Summary
  - Development Plan
  - Release Plan

- Engineering Details
  - Compilers
  - Objects & Images
  - Binary Translator
  - Early Boot Path
    - Boot Manager
    - SYSBOOT
    - Virtual Machines
    - Condition Handling
  - Dump Kernel
  - Conditional Code Verification

Project Summary - Development

The Development Plan consists of five strategic work areas for porting the operating system to the x86_64 architecture.

- VMS supports nine programming languages, six of which use a DEC-developed proprietary back-end code generator on both Alpha and Itanium. We are creating a converter to internally connect these compiler front-ends to the open source LLVM back-end 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 implement the industry standard x86_64 calling conventions and executable image format. We moved away from proprietary conventions and format 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 we are implementing a number of architecture-defined interfaces that are critical to the inner workings of the operating system.
- VMS is currently built for Alpha and Itanium from common source code. We are adding support for x86_64 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.

Project Summary - Release

The Release Plan includes multiple stages.

- Once we achieve First Boot we will work towards an Early Adopter Kit. We define First Boot as being able to bring 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 non-privileged code.
- The Early Adopter Kit is for our 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 for partners to start verifying their code on x86.
- There may be another less-than-complete system release depending on our own testing and feedback from our partners.
- The full production General Release will be the complete system as it ships today on Alpha and Itanium.

Compilers

One measure of the **GEM-to-LLVM** converter (G2L) work is the conversion of GEM tuples to LLVM's internal language constructs. For those needed by C and BLISS for First Boot we are down to the final 10% - 20%.

The C and BLISS compiler **builtins** GEM handles internally are now implemented in G2L. LLVM intrinsics are used when they can assist in a builtin's implementation. Other compiler builtins call system routines and we are starting to look at those as the relevant architecture-specific work progresses.

A system build through the compile phase was recently done for the first time with both C and BLISS compilers. This has further refined the focus on what is needed for First Boot.

The internal **MACRO** compiler structure is expanding. The basic VAX-to-x86 instruction substitution framework is in place and the register mapping implementation is being finalized, that is when you call a routine and return from it, did the correct things happen?

Objects & Images

Both the Calling Standard and ELF documents are as complete as they can be at this time. Minor changes may be needed as implementation of system components progresses but the implementations of the key system components creating and using the specifics are in progress.

As expected, a number of areas do not require much invention given our switch to industry standards in the Alpha-to-Itanium port.

The LINKER, LIBRARIAN, and Image Activator are in various stages of prototyping and implementation.

The LINKER can now take an object file produced by the x86 LLVM-based C compiler and create an x86 executable image. There is still much to be done in the LINKER but the basics are in place. ANALYZE/OBJECT and /IMAGE are complete and can be used to validate the contents of the objects and images.

The LIBRARIAN, used to create an object library, will be implemented in parallel with the LINKER.

The image activator's internal structure is starting to take shape now that the internal format of an executable image is defined.

Binary Translator

Alpha-to-x86 binary translator prototyping is underway. It parses an Alpha image, sets up simple memory mapping and starts emulating Alpha instructions. Early work includes decoding fixups, integrating an emulator, locating the image sections, and verifying the call/return interfaces.

Some code from the simh Alpha emulator is used and the protoytpe translator currently runs on linux or Windows/cygwin.

The next milestones include 1) running some simple programs all the way to completion in emulation and 2) start generating LLVM internal representation from the emulator.

The Itanium-to-x86 translator work will follow the Alpha-to-x86 work.

Early Boot Path

Reminder: The early boot path is being streamlined and modernized to be more suitable for the UEFI / ACPI environment. The functions of the former VMS primary bootstraps (VMB/APB/IPB) have been merged into the **boot manager** VMS_BOOTMGR and the VMS **loader** SYSBOOT.

We recently upgraded our sources to the latest **ACPI** Component Architecture and have incorporated the kernel pieces. Preliminary testing is done with thorough verification testing to occur as we get more of the system running. We are now bringing the user-space ACPI code analysis and system monitoring tools to VMS. These tools are new for us and will greatly enhance our ACPI capabilities.

Once the basics of local **memory disk** booting were in place we moved on to remote booting from a web server. (It happened to be a PC/Windows system but it could be anything.) The shell, the boot manager, and the memory disk file are downloaded and then the system boot is initiated.

We continue to work with **virtual machines** as well as running natively and we reach the end of SYSBOOT in all environments. In addition to creating a USB stick on VMS and taking it to an x86 system to boot natively, we can also take it to the MacBook Pro, create a virtual disk, and boot a VirtualBox virtual machine.

Early Boot Path, continued

Reminder: The x86 **SYSBOOT** code being developed is, like the boot manager, in Microsoft C, built with Windows Visual Studio, then transferred to the appropriate platform for debugging. We have also compiled the SYSBOOT code on VMS with the our new x86 LLVM-based C compiler so there are no surprises when we eventually move to building everything on VMS.

Nearly all of SYSBOOT's tasks are now written in C with just a couple MACRO-32 and native assembler modules remaining to be addressed. The initial execution pass through SYSBOOT is done; it runs to completion on multiple hardware platforms and virtual machine implementations, skipping over a few steps.

Work now focuses on filling in the skipped steps and preparing the boot environment for EXEC_INIT which SYSBOOT loads when it finishes its own work. The final steps in SYSBOOT now in progress are: setting up the boot-time I/O database, enabling boot QIOs to the memory disk, and creating the infrastructure to enable exceptions and interrupts.

Early Boot Path, continued

Reminder: Most of the code in the x86 **condition handler** EXCEPTION.EXE will be in C.

Since this code has few dependencies on the rest of the system, it presents another opportunity to take advantage of Visual Studio and the boot environment. Normally, interrupts are fielded by SWIS and sent off to the appropriate handlers, many of which go through EXCEPTION. We are “borrowing” that initial vectoring from SWIS and temporarily putting it in the new x86 EXCEPTION to facilitate some early debugging. This image can be loaded and started directly by the boot manager. With a little creativity we can force the EXCEPTION code into the various condition handlers and do a lot of debugging natively on x86 now.

Dump Kernel

Reminder: During normal system boot a second, minimalist OS instance is loaded into memory but not started. When the system goes down the primary kernel gathers and stores information; BUGCHECK then notifies the Boot Manager to boot the Dump Kernel which writes the crash dump using the runtime driver and finally initiates a shutdown.

In the beginning we did not know just how far we would need to boot the Dump Kernel but that now has settled in at the point where SYSINIT mounts the system disk.

The dump kernel is now successfully writing dumps - full and selective, raw and compressed. The dumps are being analyzed with SDA to verify their contents. (This initial testing is all being done on Itanium.) Many details remain but the major functions are solidly in place.

We previously thought the new Dump Kernel mechanism would require the crash dump file not be on the system disk, that is dump off system disk (DOSD) only. This is no longer the case due to some new insights on the timing of mounting the disk.

Conditional Code Verification

There are roughly 800 code modules needed for First Boot that contain conditionalized code, that is code compiled for one or more but not all platforms. Each of these source code modules must be checked to verify the conditionals produce the correct code for all three architectures – Alpha, Itanium and x86.

Some detailed changes will have to wait until we get further along but the vast majority of the work is easily done now. This work is underway and is proceeding well. It involves MACRO-32 source, BLISS source, C source, structure definition language (SDL) source, C header files, BLISS require files, and MACRO include files.

Note that since this work immediately becomes part of the Alpha and Itanium system builds, the conditionals affecting those two platforms get tested in our normal regression testing now.

For more information, please contact us at:

RnD@vmssoftware.com

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