vms Software

State of the Port to x86_64 January 2017

January 6 , 2017

Update Topics

- Compilers
- Objects & Images
- Early Boot Path
- Virtual Machines
- Dump Kernel
- Paravirtualization
- Condition Handling

Compilers

One measure of the **GEM-to-LLVM** converter (G2L) work is the conversion of GEM tuples to LLVM's internal language constructs. 109 have been completed and 37 remain – 12 for C and 25 for BLISS; those numbers have been determined after months of testing. The tuple conversion continues at a steady pace. This is by no means the total G2L implementation story but it is a good, general indicator of progress.

Implementation is in progress of the compiler **builtins** GEM handles internally that will be done in G2L. Other builtins call system routines which will be ported when we work on those functional areas of the system. LLVM intrinsics are used when they can assist in a builtin's implementation.

GEM and LLVM both have extensive debugging capabilities. GEM and LLVM debugging output mechanisms are being unified so that GEM, G2L, and LLVM have common output notation.

Compilers, continued

While continuing to use standard **C** language test suites we moved on to compiling individual VMS system source modules, then to the entire CRTL, and finally a full VMS system build up through the compile phase using the LLVM-based C compiler. We now have a comprehensive to-do list for what is required to build VMS itself.

BLISS will benefit from some features in newer versions of LLVM than the one we are currently using, for example implementing BLISS's explicit mapping of variables into PSECTs. We will update our LLVM, or back port the things we need, soon. We will also be moving to a wider scope of testing for BLISS once a little more G2L infrastructure is in place. A complete system build with the LLVM-based BLISS compiler is in the very near future.

The overall **MACRO** compiler structure is taking shape and experimental VAX-to-x86 instruction substitution is underway to evaluate use of registers, the stack, and memory. This will help solidify the register mapping needed for x86.

Object & Images

The Calling Standard document is now in review, pulling together everything learned to date and specifying the details needed for VMS. The starting point is the AMD64 ABI and there are a number of challenges; for example register usage, PC-relative addressing, procedure values, and the memory model, to name a few that require some design work after evaluating multiple options. This work affects the LINKER, the image activator, INSTALL, debuggers, the compilers, and any analysis tools that know the internal structure of an object, an image, or a running process.

ELF extensions will soon be added to this review.

Once the basics are finalized we can finish the remainder of the Calling Standard in parallel with continuing work on the compilers, LINKER, and tools.

We are not defining any details of the binary translator at this time but we do need a few “place holders” in the Calling Standard which can be expanded when the time comes.

Early Boot Path

The early boot path continues to be streamlined and modernized to be more suitable for the UEFI/ACPI environment. The functions of the former primary bootstrap (VMB/APB/IPB) have been merged into the Boot Manager and SYSBOOT.

We recently downloaded a new version of the industry-standard ACPI Component Architecture; this is about a nine-year update in one shot so there is verification testing of our current usage to do before moving forward. Making more comprehensive use of what ACPI provides is a major part of improving VMS device discovery and management. For example, VMS device probing is no longer needed since the ACPI tables contain the complete path to every device on the system.

The Boot Manager now recognizes just about any PCI or USB device and this allows us to interact with devices using VMS device names rather than their UEFI ‘fsx:’ equivalents. Also, using UEFI’s BlockIO drivers allows us to support a much larger number of boot devices.

Now that x86 VMS always boots from memory disk this new code gets plenty of testing from everyday development work and the details are always being fine tuned. We will soon start testing booting 1) over the network and 2) from DVD. The memory disk file itself is what gets download rather than individually loading the 100+ individual files as is the current case. The initial memory disk file is created by the VMS build/kitting process.

After a few GPT “realizations” about Windows operation we can create bootable, GPT-type USB sticks from VMS which are compatible with both VMS and Windows file systems. This eases the task of testing software that is currently developed under Windows and moved to VMS.

Early Boot Path, continued

All of the current SYSBOOT work is in Microsoft C and built with Windows Visual Studio (as is VMS_BOOTMGR.EFI). We will continue in this mode until we run into more MACRO-32 or BLISS than is worth rewriting. Still a ways into the future but EXEC_INIT will clearly be a brick wall until we have the VMS compilers.

By skipping a few things (to return to later) and by converting BLISS into C, SYSBOOT continues to move rapidly. Physical and virtual memory have been analyzed and sized, and boot-time memory allocation routines are available. Exec slicing is set up, with the huge pages available for allocation. SYS$PUBLIC_VECTORS, SYS$BASE_IMAGE, and SYS$PLATFORM_SUPPORT have been loaded and sliced, and their relocations fixed up. As built by Visual Studio, SYSBOOT itself is a PE32 image not an ELF image, so it cannot be fixed up as yet. The following additional system data structures have been allocated: balance-set slots, working-set slots, system process header, paged and nonpaged pools, the error-log buffers, the SCB, and the CPU database. Allocation from nonpaged pool is enabled. The HWRPB and SWRPB have been relocated to system space.

Next up is initialization of the CPU database. The Boot Manager already starts the secondary CPUs and leaves them in the rendezvous state so once the CPU database is established it will be a simple matter to transition the CPUs to executing some code.

Virtual Machines

This has been, and will almost certainly continue to be, an adventure. Initial work is done on HP Pro3500 Intel i3 systems. When stable up to a known point, VMS_BOOTMGR.EFI, SYSBOOT.EXE, and SYS$MD.DSK are moved to kvm, VirtualBox, and Fusion (VMware on the MAC). In most cases all three behave differently form one another **and** from the HP system. The most noticeable differences are the VMs' emulation of UEFI differ in 1) device identification, 2) the creating, saving, and restoring of environment variables (EVs), and 3) console I/O.

The device discovery work done in the Boot Manager is paying off on the virtual machines. The keyboard I/O issue is now understood and we are looking at alternative implementations. A little more experimenting and use of EVs should fall into place.

Dump Kernel

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

Currently the Dump Kernel boots and retrieves the initial state and displays the BUGCHECK code and message. It was determined early in development that the dump code needs to run in a full-blown user process so it is a user-mode program that runs in the Dump Kernel's equivalent of the STARTUP process in the primary kernel.

We are currently completing the console output of the crash data. Soon to follow are MOUNT/NOWRITE the target dump disk and locating the dump file's LBNs. Then DISMOUNT and MOUNT/FOREIGN to write the crash dump file.

This new Dump Kernel mechanism requires that the crash dump file not be on the system disk, that is dump off system disk (DOSD) only.

All of the Dump Kernel testing is being done on Itanium but we do not intend to release the Dump Kernel in the Itanium VMS product.

Paravirtualization

We are writing a Paravirtualization Strategy document to describe the overall plan for paravirtualized drivers in virtual machine environments. We need to decide which paravirtual drivers to pursue, some are obvious but others are not.

We are using the VIRTIO API for the storage driver. VIRTIO is implemented by many hypervisors. Development continues implementing the VIRTIO API in the new paravirtualized storage driver VSPDRIVER, using the existing VMS AVIO driver as a template.

The SCSI (fibre channel) IOGEN configuration code is complete.

The SCSI Port Interface (SPI) routines called by the class driver to perform SCSI functions are complete.

In progress is work on the controller and unit initialization routines to implement the VIRTIO API.

Paravirtualized drivers are not mandatory for a running guest system but the improved performance they bring is a must for the future so we are trying to get a head start in this area.

Condition Handling

The design document for the VMS Conditioning Handling Facility (CHF) on x86 is in progress. There are three main topics: the basics of the Itanium implementation, how x86 architecture differs, and how key areas of the code will change. The x86 handler will be much simpler than the Itanium handler due to a number of architectural differences.

The CHF also contains the routines for unwinding the call stack which are used by a number of system components. Unwinding is also very different on x86 than Itanium.

Most of the Itanium code is written in assembler and resides in EXCEPTION.EXE. For x86 we will rewrite as much as we can in C while retaining the overall structural flow whenever possible. There should be very few, if any, differences for callers of this code.

Software Interrupt Services (SWIS) and the CHF are closely connected and that interface is part of this investigation and design.

For more information, please contact us at:

RnD@vmssoftware.com

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