Multi-ISA Firmware Compatibility – Bringing RISC-V and IHV Ecosystems Together

Andrei Warkentin

Intel Corporation

Abstract

There are a number of challenges to a successful standards-based RISC-V ecosystem that enables PC and server-like designs. One important challenge is interoperability with the existing IHV device ecosystem. How can we bring familiar off-the-shelf PCIe devices, such as graphics, network, and storage adapters, to UEFI RISC-V systems with the same pre-boot experience seen on Intel 64 and SystemReady AArch64 platforms? This paper covers an emulation-based approach and presents MultiArchUefiPkg – an open-source solution.

Introduction

The open-source nature of RISC-V has paved the way for many companies to collaborate and define core technologies, while building products with unique differentiation and market focus. This has led to a vibrant competitive landscape that has seen RISC-V based technologies pushing into segments dominated by Arm and x64-based incumbents much faster than could have been anticipated. With less than a decade since the formation of RISC-V International, RISC-V based PCs, servers and mobile products don't seem too far off. Reinforced by an interest in sovereign compute and growing cloud provider appetites, RISC-V stands to benefit greatly from the ongoing « heterogenization » of standards-based infrastructure seen with the introduction of Arm SystemReady-compliant systems and their adoption since 2018.

After many false starts since the 90's, the Arm SystemReady ecosystem was the first to prove the benefits of a PC-like approach, rooted in multiple vendor interoperability and adoption of existing standards. Familiar hardware and firmware choices minimize friction with adopters, integrators, and implementers. This mantra of "boring hardware" lead to a set of hardware and firmware specs that distilled the x64[1] ecosystem, decoupling it from historical ISA-specific legacy. While early Arm server designs coupled custom I/O along inside the SoC, partially to differentiate and partially to compensate for weak compute offerings, subsequent wins focused on CPU, memory and PCIe performance, leaving I/O capabilities to pluggable devices. Thus, interoperability with current off-the-shelf PCIe devices remained critical – storage adapters, network controllers, GPUs, etc.

Efforts to define a standards-based RISC-V ecosystem are ongoing. Much of this work may follow in the SystemReady footsteps, building upon a decade of twists and turns of abstracting PC standards away from Intel Architecture. For example, the OS-A-SEE Task Group is currently working on the Booting and Runtime Services Specification (BRS). For interoperable designs such as servers and PCs, the BRS embraces UEFI and ACPI firmware, much like the x64 PC and Arm SystemReady ecosystems. Interoperable hardware is yet to be defined, yet interoperability with existing IHV products is equally important given the market segments such standards enable. How can a non-x64 system make use of hardware manufactured for x64 systems? Sure, the plug-in devices are all PCIe and the default server OS – Linux - is likely to have working vendor drivers, yet what about the UEFI firmware itself?

PCIe devices and UEFI

Let's assume a typical x64-based PC with a PCIe GPU. What happens when the power switch is flipped? First, the CPU boots a UEFI firmware implementation. This is a rather advanced environment, more of a minimal OS than a ROM monitor. UEFI's sole function is to initialize the minimal set of hardware to boot an OS. UEFI achieves this with, among other things, a rich device driver model with full support for plug-in PCIe adapters. PCIe busses are enumerated, with devices identified and resources configured. After this step every PCIe adapter has a "device handle" created, allowing the device to be discovered for the purpose of binding UEFI drivers to it. Drivers are either part of system firmware (e.g. standard stuff like xHCI and NVMe), or come from the adapters themselves. A driver bundle is read from the PCIe device ROM and can have multiple drivers. E.g, a video card could have an x64 UEFI driver, a legacy BIOS driver, etc. The x64 UEFI implementation will look for a x64 driver. If no supported drivers exist, the device cannot be used inside UEFI. What about a RISCV64 system? The card does not have a RISCV64 driver, so no driver would be loaded.

UEFI solved this exact problem two decades ago via EFI Byte Code. This was an abstract VM that PCI Option ROM drivers could be compiled for, that would guarantee interoperability regardless of ISA. Unfortunately, the tech was too early for its time. Itanium, where EFI made its first splash, had no interoperability concerns with BIOS-

[1] 64-bit x86, aka x86-64, AMD64, IA-32e, EM64T, Intel 64

booting PCs. When UEFI finally reached x64 PCs, no interoperability concerns existed either – legacy hardware could be supported by booting PCs in BIOS compatibility mode. It couldn't have helped that the only C compiler for EBC was a Windows-only commercial product. IHVs ignored EBC. EBC ran via an interpreter, so there were clear performance/qualification differences involved as well. By the time the Arm SystemReady ecosystem needed a solution, EBC wasn't it.

Support via Binary Translation

In 2017 Suse and Linaro engineers Alexander Graf and Ard Biesheuvel unveiled[2] X86EmulatorPkg – an open-source UEFI driver that ran x64 UEFI drivers on Arm. X86EmulatorPkg uses a binary translator from Qemu (TCG). X86EmulatorPkg models an x64 UEFI boot service environment, providing native services to x64 UEFI binaries, and vice versa. Thus, a graphics driver can provide the interface to draw to a framebuffer, and a NIC driver an interface to send and receive packets. This is possible without special-casing individual services and interfaces because UEFI exercises a quite narrow subset of an ABI: parameters are passed via integer registers, and the return values always fit within a single 64-bit register. Seamless transition from native to emulated code is achieved by mapping the x64 as non-executable, with the page protection trap handler redirecting execution to the BT engine. Transition from emulated to native code is done by comparing the branch target instruction pointer against known ranges of emulated images, performing native jumps instead of going through the BT layer. For driver I/O, x64 port I/O instructions are mapped to UEFI PCI I/O operations, but very little else is done with respect to defining a minimal well-defined x64 UEFI Boot Service environment. In practice, X86EmulatorPkg does a great job of supporting current GPUs and NICs on real production Arm servers.

MultiArchUefiPkg for RISC-V

An effort to bring-up X86EmulatorPkg on RISCV64 started in late 2022, as part of creating more realistic and useful reference RISC-V platforms to demonstrate the value of interoperable standards. In April 2023, the resulting work had been released as open source to the general community for further collaboration as **MultiArchUefiPkg**[3]. Like X86EmulatorPkg, MultiArchUefiPkg models a foreign ISA UEFI boot service environment, fit for running well-written UEFI applications and device drivers.

X86EmulatorPkg is very tightly coupled to portions of Qemu TCG code, which at the time had no RISC-V support, and which were difficult to identify as belonging to any specific version to help with a possible rebase to newer Qemu bits. Instead, a decision was made to rewrite X86EmulatorPkg using the Unicorn Engine, which is an open-source CPU emulator library also based on Qemu. The full rewrite, known as MultiArchUefiPkg. decouples the UEFI emulation driver from the CPU emulation library using well-defined API. The net result has competitive performance, portability, support for multiple emulated ISAs (x64 and AArch64 on RISCV64), size (2/3 the binary size on AArch64 compared to X86EmulatorPkg) and improved correctness in modeling the emulated environment, such as handling of native LongJumps, image exits and self-modifying code. A regression test application exists for basic smoke testing of changes. Integration with Unicorn Engine itself brought some challenges. The libraries had to be ported to the UEFI environment, and several fixes were necessary for correctness and performance. The modified Unicorn Engine sources are also released[4] with intent to upstream.

Architecture-specific differences between AArch64 and RISC-V support are minor, mostly adding code to recover the x64 RIP from the exception program counter (SEPC) and writing the assembler level shim to convert exception state to CPU emulator call arguments.

The project has been a great test for RISC-V UEFI implementations readiness and spec bindings. For example, bugs in the UEFI RISC-V timer driver and exception handling code were fixed and contributed back, and various gaps are yet to be addressed via UEFI spec ECRs. More crucially, existing RISC-V UEFI implementations were found to have no support for MMU page protection. An MMU support patch set from Ventana Micro Systems should be merged soon, but as a stop-gap this prompted some further innovation around the use of native EFI wrappers around emulated image entry points.

Future

Binary translation is a great tactical solution to bring the existing IHV ecosystem to RISC-V, but long term it's not a great solution without further efforts to narrow down the subset of allowed behaviors in the emulated code, perhaps via a custom toolchain. For example, the use of newer ISA extensions in code (some new revision to SSE, perhaps) would mean more changes and complexity. Even today, for example, some EFI applications don't run, expecting support for invariant RDTSC instead of using UEFI services. On the other hand, expecting IHVs to bundle multiple ISA support doesn't seem like a great strategy either with at least 4 modern 64-bit architectures with UEFI support now in the wild, and 128-bit ISAs around the corner. Another approach could be to resurrect EBC, addressing tooling and performance concerns, or to investigate something similar yet more modern like WebAssembly.

[2]http://events17.linuxfoundation.org/sites/events/files/slides/QEMU%20in%20UEFI.pdf

[3] https://github.com/intel/MultiArchUefiPkg

[4] https://github.com/intel/unicorn-for-efi