Beyond Virtualization: A Better Approach to Multi-Core Resource Allocation and Control with Linux

It is a widely held misconception that in order to fully leverage the high performance available with multi-core processors, virtualization in the form of a hypervisor must be used, along with a combination of Linux and either an RTOS or simple runtime environment. Fueling this misconception is the notion that Linux itself is incapable of meeting the requirements by itself, because it is some combination of too big, too slow and not real-time. Often times it is RTOS vendors who perpetuate this virtualization misconception.

These misconceptions about Linux and virtualization drive added complexity and costs into the development process. Why? Complexity increases due to multiple run-time and development environments. Costs increase because of likely royalties for the proprietary RTOS and hypervisor, not to mention the added costs created by the development complexity itself.

By using known technologies, you can deliver a highly configurable, scalable, and virtualized Linux environment that includes a very low overhead run-time capability that can match bare-machine and/or RTOS performance. This approach can significantly simplify the development of multi-core applications by eliminating the need to deploy multiple run-time technologies, ultimately lowering the overall cost of development.

Virtualization Overview

Virtualization can be described as a method for dividing the resources of a computer into multiple execution environments. There are three major categories of virtualization in use today, the key difference among them being the layer where the virtualization occurs.

- Full virtualization - Slowest method, but does not require changes to the OS or applications. Virtualization is done transparently at the hardware level of the system. QEMU and KVM are two examples of full virtualization.

- Paravirtualization - Faster than full virtualization but requires changes to the OS and possibly the applications to take full advantage of optimizations of the virtualized hardware layer. Xen offers a paravirtualization solution, but requires dedicated hardware assist to run un-modified guest operating systems.

- OS resource virtualization - Fastest method (less than <=1% overhead) and requires no changes to applications since the virtualization is at the OS API level. By creating a virtualization layer for OS resources, many applications can run on a single host OS while maintaining the illusion of having the host all to themselves. BSD Jails and Linux Containers are examples of OS Resource Virtualization.

These various types of virtualization offer different performance characteristics, require different setup and maintenance overhead, introduce unique levels of complexity into the run-time environment, and address different problems. While the industry is currently focused on pushing fully virtualized hypervisors as the one-size-fits-all solution to multi-core optimization,

embedded developers need a range of options that can be tailored to specific application needs. Developers will require some combination of one or more of the virtualization technologies listed above to deliver products that fit within hardware constraints and meet design performance characteristics.

An Alternative Approach

A hypervisor is simply one approach for delivering virtualization in a Linux environment. There are other options available, all based on non-proprietary, open source Linux technology. These options are:

- KVM Hypervisor
- Linux Containers
- Bare Metal Engine™

KVM (Kernel Based Virtual Machine)

KVM (Kernel-based Virtual Machine) is a hypervisor-based full virtualization solution for Linux on x86 hardware. Ports are under development for MIPS, ARM and PowerPC architectures.

Using KVM, one can run multiple virtual machines running unmodified Windows (on x86 only of course) or Linux images (where there will be support for both Linux and non-Linux operating systems native to the underlying processor architecture). Each virtual machine has private virtualized hardware. For embedded SoCs, each implementation of KVM needs to determine which of the SoC-specific hardware devices will be incorporated into the KVM environment. The kernel component of KVM has been mainline Linux as of revision 2.6.20.

Linux Containers

Containers implement virtualization of operating system resources. The virtualization layer for Containers operates on top of a single instance of Linux giving each Container the illusion of controlling system resources, even though they may be sharing those resources with other processes or Containers. Linux can partition resources, such as CPU bandwidth and memory, to balance the conflicting demands of multiple Containers.

As an OS-level virtual environment, Containers provide lightweight virtualization that isolates processes and resources without the complexities and overhead of full virtualization. Containers provide a virtualized environment that supports multiple process and network name space instances as well as extensive resource allocation and control capabilities.

Types of Containers

Linux Containers are used in two ways:

Application Workload Isolation — Application Containers provide the ability to run an application on the host OS kernel with restrictions on resource usage and service access. Application Containers run with the same root file system as the host OS, although the file

system namespace may have additional private mount points. Access to resources, such as memory, CPU time, and CPU affinity, can be limited. A private network namespace can be set up to restrict connectivity.

Completely Virtualized OS — System Containers provide a completely virtualized operating system in the Container. The Container has a private root file system, a private networking namespace, additional namespace isolation, and configurable resource restrictions. A system Container starts at the same place as a regular Linux distribution: it starts by running init.

Containers can share parts of the root file system with the host operating system or other Containers. Sharing portions of a root file system is an excellent way to conserve disk space. Generic binaries and libraries are commonly shared between Containers. The integrity of the file system being shared can be preserved by using read-only bind mounts.

Advantages and Disadvantages of Containers

Containers offer several advantages over full virtualization:

- Reduced Overhead — Containers impose little overhead because they use the normal system call interface of the operating system and do not need emulation support from an intermediate-level virtual machine.

- Increased Density — More useful work can be done by applications because fewer resources are consumed by the complexity of full virtualization. Given the same machine, you can run more Containers on it than virtual machines.

- Reduced Sprawl — Maintaining multiple different operating systems in a full virtualization environment can be a hassle. Because Containers can share many resources with the host OS, upgrades and modifications to the underlying operating system propagate seamlessly to any Containers sharing the underlying file system.

There are however, some drawbacks to Containers:

- Reduced Flexibility — Operating system-level virtualization is not as flexible as other virtualization approaches because it cannot host a guest operating system different from the host, or a different guest kernel. The requirement for multiple, disparate, operating systems would be a use case for full virtualization.

- Decreased Isolation — Because the kernel of the underlying operating system is shared between Containers, there is less isolation than with full virtualization.

Containers Use-Cases

Containers may be used for a number of different use cases. These include:

- Resource partitioning with maximum performance
- Multiple secure applications instances (e.g. a walled garden)
- Process isolation (e.g. process jails)

- GPL insulation

Bare Metal Engine™

It is possible to utilize elements of Container technology to optimize performance in a multi-core environment. In particular the power of the resource allocation and control capabilities of Containers can provide a very low-overhead execution environment suitable for high-throughput packet handling and other performance-critical applications.

The conventional wisdom is that there is "lots" of overhead in Linux precluding its use in performance-critical applications, such as network packet processing. However, if we could enhance Linux to operate on some cores at the same level of efficiency as that supplied by a typical RTOS we could eliminate the need for a separate RTOS development environment.

MontaVista achieved the near "zero overhead" goal for a dedicated core(s) with a new capability **Bare Metal Engine™** (BME).

BME is a configuration of several elements of Linux technology, including virtualization, to optimize performance in a multi-core environment. In particular BME uses the power of the resource allocation and control capabilities of Containers (OS-level virtualization) along with other optimizations to provide a very low-overhead execution environment suitable for high-throughput packet handling and other I/O intensive embedded applications, using only the well understood Linux application programming model.

Today, we program performance-critical, I/O-intensive applications in Linux kernel mode. Kernel mode applications have complete access to hardware and Linux internals and make a tempting choice for programmers reaching for the highest possible hardware performance. But the kernel mode environment is also complex, inflexible (no C++ for example), unstable (interfaces can change release to release) and typically not well understood by application programmers. A simple programming error in kernel mode can be fatal to the system.

But we can we achieve the performance we require and operate in the more secure user-mode environment. Using a multi-core SOC (OCTEON II™ with 32 cores for example), we run MontaVista Linux CGE with SMP enabled. We use processor affinity to dedicate a User-Mode application process to a core. We route only one interrupt to that core. We use only the Linux UIO (*User Mode Driver)* interface to handle and synchronize this interrupt with its associated User-Mode application process. We enable tickless Linux capability. We make sure that the process' address space is mapped such that there is no paging and that its memory can be mapped completely within the TLBs available. We also arrange that I/O registers and DMA memory of the accelerator hardware are mapped into our space. We use NPTL for multi-threading within the process.

We have now configured access and control of the accelerator hardware and confined the environment to user-mode. By this approach we have achieved our goal: high packet performance without resorting to burying the software inside the Linux kernel.

Summary

By using the techniques described, application developers can now deploy Linux across all of the cores on a multi-core processor and avoid the complications of multiple run-times (e.g., Linux, an RTOS and a hypervisor). The end result, a highly configurable, scalable, and virtualized Linux environment that includes a very low overhead run-time capability that can match bare-machine and/or RTOS performance, with a lower cost of development.