WHITE PAPER

BEYOND VIRTUALIZATION

The MontaVista Approach to Multi-core SoC Resource Allocation and Control

ABSTRACT

MontaVista Linux Carrier Grade Edition (CGE) provides a Linux-based programming environment that can scale from high-performance "bare metal" dedicated spaces, to an RTOS-like multi-threaded run-time, up to a fully virtualized Linux SMP process model. Application developers can now deploy a single operating system, 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).

This paper outlines that approach to 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 level performance in Linux.

Table of Contents

INTRODUCTION

It is a widely held misconception that in order to fully utilize the high performance available with multi-core processors a combination of Linux and either an RTOS or simple runtime environment must be utilized. Fueling this misconception is the thought that Linux itself is incapable of meeting the requirements alone, because it is some combination of too big, too slow, and not real-time. This misconception also drives the requirement that hypervisors and/or virtualization are required to mediate and isolate the different run-time environments and facilitate the inter- communication among them. Often it is the RTOS vendors themselves who perpetuate this misconception.

In the end, these misconceptions about Linux drive added complexity and costs into the development process. Complexity increases due to multiple run-time and development environments (one each for Linux, the RTOS and possibly the hypervisor.) Costs increase because of likely royalties for the proprietary RTOS and hypervisor, not to mention the added costs created by the development complexity itself, e.g., more developers for a longer period of time.

This paper outlines an approach to using a number of known Linux technologies to 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. With MontaVista Linux Carrier Grade Edition 6, Linux alone is able to support all of the operating system performance and functionality requirements required by a modern multi-core application.

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 the three types of virtualization is the layer where the virtualization occurs. See Figure 1 for an overall view of virtualization technologies and their relative performance/isolation tradeoffs.

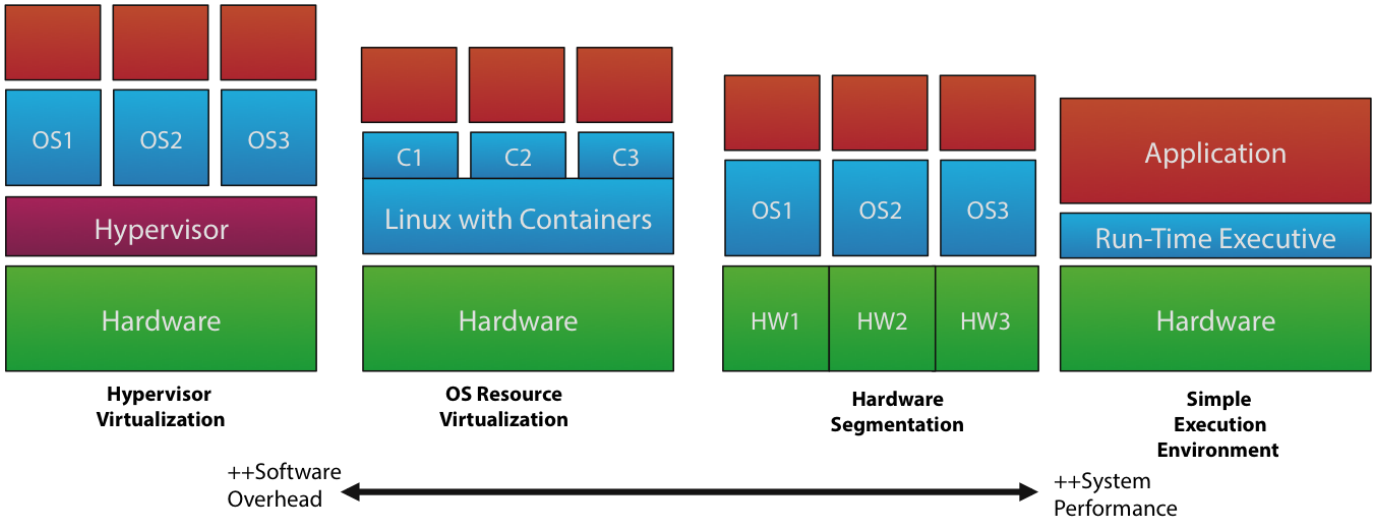
- Full virtualization - Slowest method, but does not require changes to the OS or applications. Virtualization is done transparently at the hardware level of the system. QEMU and KVM are two examples of full virtualization.

- 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 for improved performance. 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. Operating systems manage a number of resources that must be shared across applications (e.g. network ports). 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.

Hardware segmentation has traditionally been available in mainframes and high performance computing (HPC) where large, complicated systems were segmented by function to allow multiple simultaneous users to run in dedicated environments. Multi-core chip vendors are now providing similar functionality on complex SOCs, where the hardware itself can be segmented into many individual systems, with shared and dedicated resources



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.

THE MONTAVISTA APPROACH

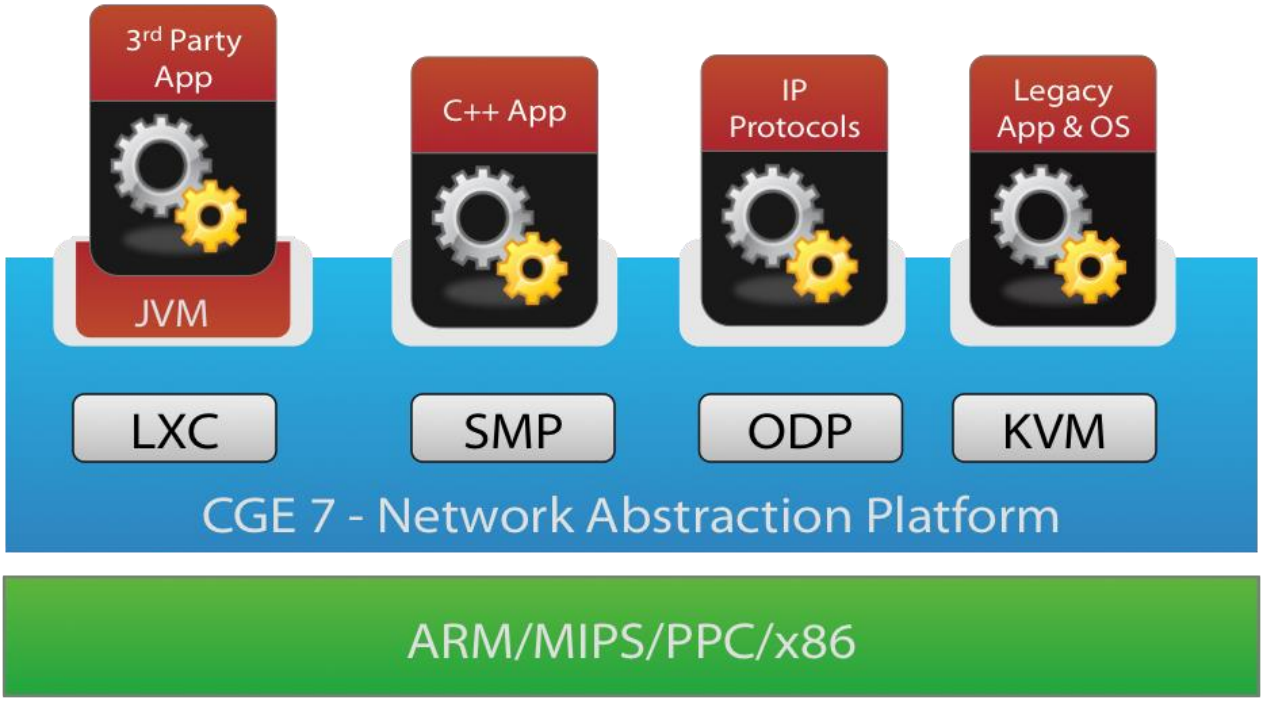
MontaVista provides three methods of virtualization to offer the best combination of flexibility, performance, and ease of application development. All are based on non-proprietary, open-source Linux technology and are supported by MontaVista across multiple processor architectures. (See Figure 2 for an overall picture of the MontaVista approach.) These three methods are:

- KVM Hypervisor (full virtualization)

- Linux Containers (OS resource virtualization)
- MontaVista Bare Metal EngineTM (OS resource virtualization & SoC hardware optimizations)

KVM (Kernel Based Virtual Machine)

KVM (Kernel-based Virtual Machine)[1] is a hypervisor-based full virtualization solution for Linux on x86 hardware (both Intel and AMD variants) with ports under development for MIPS, ARM and PowerPC architectures. It consists of a loadable kernel module that provides the core virtualization infrastructure and a processor-specific module for Intel or AMD processors, and the equivalent for other architectures. KVM also uses a modified QEMU[2] to support the I/O requirements of the virtual environment.



Using KVM, one can run multiple virtual machines hosting unmodified Windows (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: a network card, disk, graphics adapter, etc. 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.

The following is a brief summary of KVM support:

- Intel-based hosts (requires VT capable processors)
- AMD-based hosts (requires SVM capable processors)
- MIPS, PPC, and ARM architectures under development
- Windows/Linux/Unix guests (32-bit and 64-bit versions)
- SMP hosts and SMP guests (as of kvm-61, a maximum of 16 CPUs supported
- Live migration of guests from one host to another (both 32-bit and 64-bit versions)

See http://www.linux-kvm.org/page/Guest_Support_Status for a list of guest operating systems known to work with KVM. For information on supported host hardware see http://www.linux-kvm.org/page/Host_Support_Status.

Since paravirtualization of I/O devices can significantly increase the performance of virtualized systems, there is considerable effort in the community to improve support[34].

There are three primary use-case scenarios we think are the most likely to be used by CGE 6 developers:

- CGE 7 running as a guest OS on KVM
- Earlier versions of MONTAVISTA CGE
- Customer legacy applications and the underlying RTOS

Allowing CGE 7 to run as a guest within CGE 7 KVM is a standard supported configuration of CGE 7. Both earlier versions of CGE and customer legacy applications using KVM can be supported configurations if the MontaVista Virtualization Services have been purchased. Contact MontaVista for further details on this as well as other options for KVM support.

CONTAINERS OVERVIEW

Containers implement virtualization of operating system resources. The virtualization layer for Containers operates on top of a single instance of the host OS, in this case the Linux operating system. The host OS gives each Container the illusion of controlling system resources, even though they may be sharing those resources with other processes or Containers. The host OS can partition resources, such as CPU bandwidth and memory, to balance the conflicting demands of multiple Containers. Containers are built on top of two Linux technologies, control groups and namespaces.

There are a number of Container implementations in existence for Linux; Linux Vserver, Virtuozzo and LXC (LinuX Containers). MontaVista has chosen to commercialize LXC because it is an active project, it is well integrated into the Linux mainstream and it is supported by a capable library and tool environment. Throughout this paper, when we refer to Containers we mean our commercial implementation of LXC specifically.

---

[1] http://www.linux-kvm.org/page/Main_Page

[2] http://wiki.qemu.org/Main_Page

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

CONTROL GROUPS

Access to resources for a Container can be controlled with Control Groups (cgroups). cgroups associate a group of processes with a set of parameters for one or more "resource controllers." For instance, the "cpuset" or the "sched" controllers which are part of the larger cgroup framework can be used to set the scheduling boundaries for processes in a container; the "memory" resource controller can be used to control the amount of memory assigned to a container; or the "network" controller can associate a network class identifier with each container that can be used to classify packets to implement QoS scenarios.

cgroups are organized in a virtual file system that is normally mounted under /cgroups. Mounting the cgroup file system creates a hierarchy of all registered subsystems.

Subsystems are kernel modules that are designed to control a specific resource. They are used to allocate varying levels of system resources to different control groups. There are seven subsystems:

- cpuset—Assigns CPUs and memory nodes to a control group.
- cpu—Provisions the access to the CPU.
- cpuacct—Reports CPU usage.
- memory—Sets memory usage limits and reports memory usage.
- devices—Sets access policy to devices.
- freezer—Suspends and resumes the tasks in a control group.
- net_cls—Tags network packets with a class identifier so that traffic can be shaped, scheduled, policed or dropped.

Subsystem parameters for a control group are represented as pseudo-files in the cgroup virtual file system. These pseudo-files can be manipulated with shell commands or system calls. You can query and set values by reading and writing to the various pseudo-files.

NAMESPACES

Namespaces provide resource isolation for implementing Containers. A container is essentially a group of processes, with access to a subset of system resources virtualized by cloning resource namespaces. Linux system resources such as process IDs, IPC keys, or network interface identifiers have traditionally been identified in global tables. The namespaces feature transforms these global resource identifier tables into tables (namespaces) specific to groups of processes. The ability to create multiple instances of a namespace enables multiple resources (processes, users, network interfaces etc) with the same identifier, within a single instance of the Linux kernel.

---

[3] http://www.linuxkvm.org/page/Paravirtualized_networking

[4] http://www.linukvm.org/page/Paravirtualized_block_device

Example namespaces in the Linux kernel are PID namespace, Network namespace, proc namespace, UTS namespace, etc. The following namespaces are supported:

- UTS Namespace—UTS namespaces associate Containers with different host names. The uname() system call prints out information about the current platform. uname() is modified to provide different information depending on the namespace from where it is called. The utsname struct in glibc identifies a platform, and contains fields such as the host name, the domain name, the name of the operating system, the version of the operating system, etc. A process can request a cloned copy of utsname and make changes to it with calls to setdomainname and sethostname.
- IPC Namespace—IPC ids (semaphores, shared memory, message objects) are overloaded so that they correspond to per-namespace IPC objects.
- PID Namespace—PID Namespaces let multiple processes have the same PID, as long as they are in different namespaces.
- User Namespace—User Namespaces enable per-Container user information (uids, gids, user_struct accounting).
- proc Namespace—proc Namespaces let each Container have its own instance of the /proc file system.
- Read-Only Bind Mounts—Read-only bind mounts enable a read-only view into a read-write file system. It provides additional security when you are sharing the underlying file system with a Container.
- Network namespaces-- allow you to create per-Container instances of the network stack. Network namespaces virtualize access to network resources (interfaces, port numbers, etc.), giving each Container only the network access it needs. The network namespace is a private set of network resources assigned to one or several processes. A network namespace has its own set of network devices, IP addresses, routes, sockets, etc. Processes outside of the namespace cannot access these network resources. Processes inside the network namespaces do not know anything about the network resources outside the namespace. Resources inside one namespace do not conflict with resources inside another namespace. For example, web servers running in different network namespaces can all listen on their own port 8o.

USES FOR 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 may be used for a number of different use cases. These include

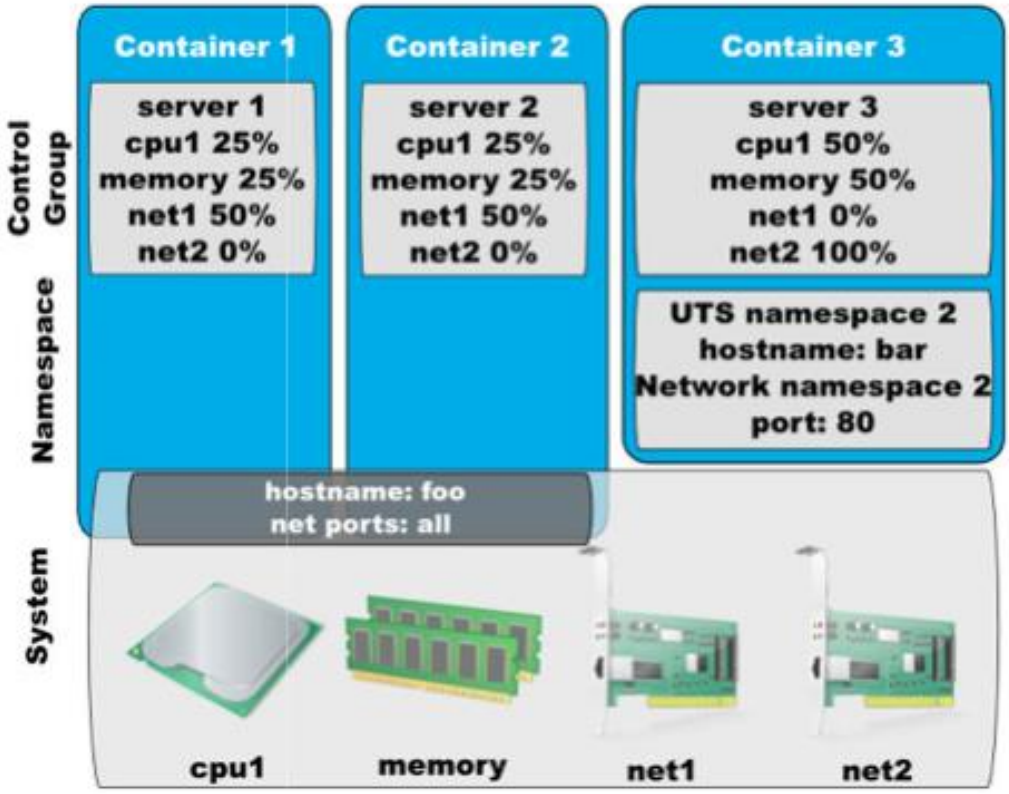


Figure 3 Container Components

ADVANTAGES AND DISADVANTAGES OF CONTAINERS

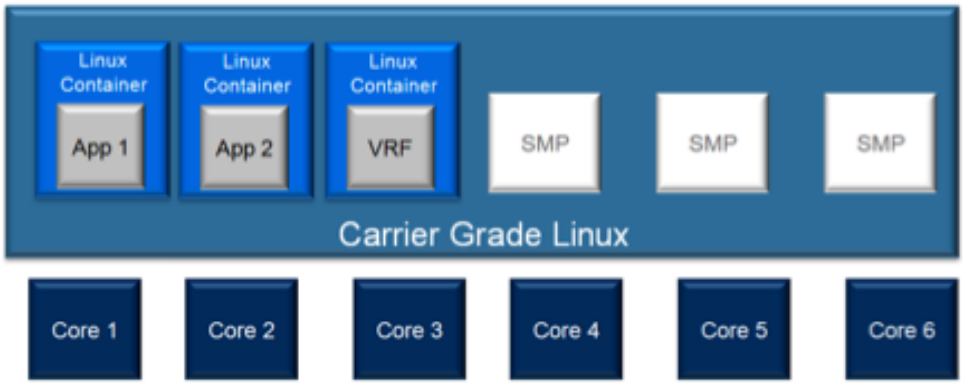
Containers offer several advantages over full virtualization:

- 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 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. For example, you cannot have a Windows Container, you can only have Linux Containers. The requirement for multiple, disparate, operating systems would be a use case for full virtualization, e.g., KVM.
- Decreased Isolation — Because the kernel of the underlying operating system is shared



between Containers, there is less isolation than with full virtualization.

CONTAINER EXAMPLE: RESOURCE ISOLATION FOR NETWORKING

Network access is a critical requirement of many systems, and Containers offer many possible network configuration possibilities. The simplest option is to use Virtual Ethernet (veth). Veth creates a new network stack and a pair of devices, one which is assigned to the Container and the other which is assigned to the host. The virtual device is added to a bridge and can be managed like any other device. Using a bridge and virtual devices in this way allows a single physical

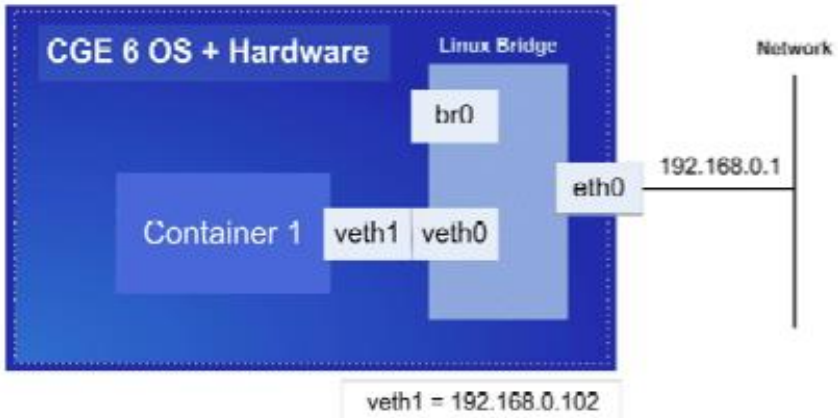


Figure 5 Configuring networking in a new network namespace

network connection to be shared with many containers.

MONTAVISTA BARE METAL ENGINE ™

Some elements of Containers can be used 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 dedicated performance-critical applications, such as high-throughput packet handlers. With the release of CGE 7, we are pleased to report that we have achieved the goal of near "zero overhead" for one or more dedicated core(s) running within MontaVista Linux CGE 7. We call this new Linux capability Bare Metal EngineTM (BME) and it is unique to MontaVista Linux.

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.

BARE METAL ENGINE EXAMPLE: PACKET HANDLING

Consider this packet-handling situation: Suppose we have a need to encrypt a data stream of 1k packets. Doing the encryption on a core with no hardware acceleration takes 100k cycles per 1k data, and per packet overhead is 500 cycles. Then on a 1 Gcps (Giga cycles per second) CPU, where the application can get 98% of the CPU cycles, it can encrypt 9751 packets per second. If, through tuning, the application can squeeze out another 1% of the available CPU cycles, then it can encrypt 9850 packets per second. But if it can get access to some encryption acceleration hardware owned by the kernel, and it takes 50,000 cycles to do the encryption, plus 10,000 combined user/kernel cycles to set up the operation and another 20 cycles per 8 bytes to copy the data into the kernel and the same to copy the data out of the kernel, then the encryption rate is 15,356 packets per second, and the application no longer needs 99% of the CPU to do it--only <24%. The application can get another 7% performance by putting the copy operations under the wait for the price of 2% additional CPU use. Further, it is possible to get that back if the copy operation can be eliminated as well. This example clearly illustrates how critical it is to have access to underlying acceleration hardware. It not only dramatically speeds up the specific operation (encryption in this example), but also significantly reduces the load on the general-purpose processor.

How do we program performance-critical, I/O-intensive applications using Linux? Typically, programmers write these applications in Linux kernel mode, which is powerful to be sure. Kernel-mode applications have unfettered access to hardware (the crypto accelerators in this example) and Linux internals, and are therefore 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. Also, a simple programming error in kernel mode can be fatal to the system. On the other hand, the most flexible, stable, safest and best-understood Linux

environment is the user-mode application programming model. Ideally, we would like to have our software cake and eat it too: the ability to have a user-mode programming environment with kernel-mode performance.

Can we achieve the performance we require and at the same time operate in the more secure user-mode environment? The answer is Yes. Here's how we do it: Imagine a multi-core SOC such as an OCTEONTM II with 32 cores. We run MontaVista Linux Carrier Grade Edition 6 for OCTEON II with SMP enabled. We use processor affinity to dedicate a user-mode application process to a particular 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 at the same time confined the environment to user-mode. By this approach, we have achieved our software cake goal: high packet performance without resorting to burying the software inside the Linux kernel.

BARE METNAL ENGINE CONFIGURATIONS

Bare Metal Engine Configurations

The base configuration uses the Linux user-mode I/O capabilities along with a suitably configured kernel to run a simple device driver and an associated application. This is intended to be a highly portable capability that allows for the driver and application to be moved around in a Linux system, including running in a hypervisor or Containerized virtual machine. These capabilities available are as for any UID = 0 process. The base configuration is a standard supported CGE 7 capability

We expect that customers desiring the highest possible performance will require more hardware-dependent configurations which provide maximum utilization of the hardware. Depending on hardware capabilities, the software may run at a privilege higher than problem-space, up to and including the kernel or hypervisor privilege. If memory mapping/protection is used, it is used in a manner allowing permitted access to resources to occur with no more translation overhead than experienced by the kernel. The intention is to provide maximum performance of the driver and associated application, approaching that available from a naked hardware implementation, at least for the fast path.

We fully expect that additional configurations will be identified as we gain experience with a growing number of supported hardware platforms. As a general rule, we will develop as much of a standard, hardware-independent API for BME as possible. But the simple fact is that there are significant differences among the vendor-specific SoC- supplied acceleration hardware

capabilities. Therefore, while many of the interfaces BME uses will be available across architectures, it is inevitable that some hardware dependencies will result from the requirement to maximize performance on any given platform. These customer-specific configurations can be developed as part of a professional services engagement. Contact MontaVista for more details.

BARE METAL ENGINE DETAILS

The Bare Metal Engine Base configuration runs with a fully translated memory space, using the normal memory and I/O access as allowed via the standard Linux user libraries and the UMIO extension. OS extensions may be considered to ensure that contiguous physical memory can be allocated to facilitate DMA buffer allocations. Standard per-process protection models are assumed.

The process/driver threads may be configured to be locked to a single processor, and any associated interrupt also may be locked to the same processor. The effect of locking onto a single core gives significantly higher performance than the performance obtained with a general SMP model, where execution and interrupts are processed by any available core.

The process runs in problem space with the privileges granted to a UID=0 process. Access to higher privilege is obtained through the normal syscall mechanism, with the attendant context switch penalties. In this manner, essentially all of the user space resources and libraries are available with the expected overheads. The actual fastpath of the process will not use those facilities unless on an exception or initialization basis.

EXAMPLE: HARDWARE OPTIMIZAED CONFIGURATION OF BME

A hardware-optimized BME configuration runs with a flat memory model, where needed I/O and memory are accessible at all times. There exists a simple, fast translation between virtual addresses and physical addresses that can be performed without consulting page tables or other translation mechanisms. The address space may be virtual or not, as allowed or required by the hardware model. The needed and allocated resources may be allocated at initialization, but thereafter, memory is managed by the process or by hardware used by the process.

Address space access protections may be used, provided the required translation and protection mechanisms are not a significant source of execution time variation. Depending on the available hardware capabilities, one or more of the multiple cores may have shared access to certain physical address regions. For example, the OCTEON II allows for virtualization of the FPA and SSO facilities, enabling resources to be isolated between the virtual machines.

Resource allocations are made either at load time or process-initialization time. This close-coupled use case allows for the process/driver to assume that only the single core has access to the required resources. While that may not actually be true due to hardware limitations, the

model can be "enforced" by a resource allocation supervisor, or by controlling the build and link process.

Single-core affinity is assumed for all of the process threads and driver and device interrupts. Most hardware can be configured for this affinity, but a small 1st level ISR may be required for certain architectures.

Thread scheduling is the responsibility of the application. Cache and TLB locking is available. Thread synchronization is also the responsibility of the application. Privileged instructions may be available if the application runs at a suitable hardware privilege level. Obviously, running in this environment places requirements on the application to behave.

Any core dedicated to this environment is unavailable to the system for other work. Certain libraries may be available, along with certain other OS interfaces. However, this is not intended to be very portable unless low- overhead abstractions can be found. That is, what runs on one chip model may not run on another without porting work.

SUMMARY

When all these technologies are integrated into a cohesive package, we are able to provide, within the Linux context, a programming environment that can scale from a high-performance "bare metal" dedicated spaces, to an RTOS-like multi-threaded run-time, up to a fully virtualized Linux SMP process model.

The benefits to developers are clear:

Lowered Complexity

The application developers now can build upon a single OS—Linux—across all of the cores on a multi- core processor and avoid the complications of multiple run-times (Linux plus an RTOS) or even an even more complex situation, such as Linux, an RTOS and a hypervisor.

Flexibility in Development

A complete hypervisor may be implemented when the use case demands it, but developers aren't forced to incur the overhead of a hypervisor when all they need is multi-core resource management using Containers or BME.

High Performance

With BME, developers can get the performance of an RTOS, without the cost and overhead of a second OS. They can deliver bare metal performance in Linux, without

MontaVista Software, LLC
2315 North 1st Street, 4th Floor
San Jose, CA, 95131
Tel : (408) 943-4500
Fax : (408) 943-7451
Email: sales@mvista.com