软件愈加多样性对嵌入式系统的影响

Maarten Koning
技术办公室
风河
美国阿拉米达
Maarten.koning@windriver.com

摘要—*复杂嵌入式系统的软件组成，如汽车、机器人和医疗设备，正经历着巨大变化。这些设备中的应用程序越来越多地与其定制操作系统实例集成、封装在一起。这一趋势始于IT域，IT应用程序率先利用虚拟机和容器；如今，嵌入式系统领域也显现了这一趋势。但是，这种变化对嵌入式系统构建方式的动机和影响却是截然不同的。随着具有不同核心价值和需求的离散多核嵌入式系统搭载越来越多的软件，一个操作系统实例甚至有着多个实例的操作系统都无法为多应用程序运行提供最佳的运行环境。可针对特定操作系统（如Linux）或特定商用RTOS开发应用程序，且可分别与开源和认证系统一起不断发展或淘汰某些应用程序。这些应用程序也可能有多个来源，因此出于支持、许可或发布间隔等原因需要对其进行分区。本文探讨了一种全新的异构分布式系统架构，该架构应用复杂而强大的多核SoC嵌入式系统硬件；还探讨了这对于创造、交付和获取嵌入式软件价值的公司的意义与影响。*

关键词—异构系统，RTOS，多核技术，虚拟化，软件分区，Hypervisor，安全性

I. 引言

智能系统中的软件内容数量呈指数增长，抵消了摩尔定律提出的晶体管增速。对于为嵌入式系统设计的硬件而言，这些晶体管用于额外的内核处理、缓存和其他内存技术以及IO——如今都位于单个多核封装中。包含此类完整计算机实现的系统统称为系统级芯片（System-on-Chip，简称SoC）。过去，离散多核处理器更为通用，且可适配SMP操作系统，以管理、调解跨多个应用程序的硬件资源。尽管概念简单，但类似方法并不总可以扩展至如今用于嵌入式系统的异构多核复杂SoC。目前尚有许多软件工程挑战；随着专用嵌入式系统软件内容数量的增加，挑战日益棘手。

在软件开发阶段和系统运行期间，若多个程序或应用程序需要运行于单个操作系统中，使用单一SMP　OS的方法就会存在诸多挑战。在软件开发阶段，这些应用程序需要迁移到同类SMP OS [1]。同时，无论是文件系统、IPC机制，亦或是网络堆栈等空间，这些应用程序都需要避免在操作系统域名空间中出现冲突。系统运行期间，受操作系统调度、内存或磁盘碎片、缓存或TLB冲突以及其他资源干扰，可能会出现拒绝服务等问题。

此外，应用程序的生命周期与单一操作系统的生命周期息息相关，因为重启操作系统时，无论应用程序的重要性如何，所有应用程序也都必须重新启动。由于安全补丁或系统软件优化更新，或硬件故障（如内存故障），正常系统操作期间也可能会发生此类操作系统重启。若将此视为系统事件，则所有应用程序都与单一操作系统的整体生命周期相关，这与系统架构师的期望相悖，他们期望较小的故障域而非全局故障域。

单一操作系统方法使得信息安全与功能安全也面临挑战。若对托管信息安全和/或功能安全应用程序的操作系统有严格要求，那么这些要求会向下映射回溯至操作系统，因为不安全的操作系统本身就会导致应用程序故障以及应用程序攻击。即使操作系统的功能和信息安全级别满足应用程序的要求，代价也会更高，因为必须将操作系统信息安全和/或功能安全范围中的所有代码（甚至是功能安全或信息安全应用程序未使用的功能代码）写入相同的安全级别-。这与关注问题隔离、特权最小化和故障域最小化的设计最佳实践相背离。

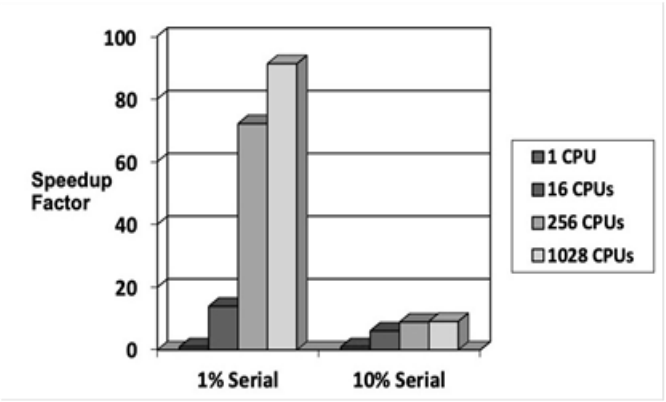
[1] 若可能，将Linux中应用程序标准化为SMP OS的趋势，在某种程度上缓解了这一挑战。

SMP OS

SMP OS

SMP OS

SMP OS



1. 阿姆达尔定律

——

II.

= 1 /

+ 1- / N N

0-1 0 1

SMP OS

SMP

1 混合关键性 2 防止故障传播 3 保证不同OS各自的核心优势

4 分离特权 5 资源调配 6 实时软件更新

1

SMP OS

工作负载合并

SoC 2 软件重用

3 组织所有权

IT

4 网络安全功能或系统管理功能

/

体系结构清晰、操作灵活

CPU

“ 集成平台 ”

SoC

OEM已经为平台开发人员、应用程序开发人员和系统集成商

III. 分区技术

应用程序分区技术可由弱分区到强分区来阐述。传统Unix风格的进程可用于承载多个独立应用程序，每个应用程序均由多个相互不干扰的活动实体（线程和进程）组成。这是通过“设计”先验AKA“白板分区”（whiteboard partitioning）完成的。但Unix风格操作系统提供的分区技术被视是“弱分区”，因为该操作系统并不强制应用程序互不干扰，反而在很大程度上都是约定俗成的，其中包括一些通用用户ID和GroupID的特定配置。

容器技术（如Docker和LXC等）利用的Linux控制组和域名空间实现“软分区”，通过提供操作系统虚拟化的主操作系统，软分区间存在干扰通道。随着更多基于硬件的功能可在硅片中实现，诸如KVM等Type 2（基于托管操作系统）hypervisor提供的硬件虚拟化也在稳步改善。通常将其称为“硬件虚拟化”或“虚拟分区”。 Type 1 hypervisor（直接运行于硬件，可支持硬件虚拟化）可实现“强分区”。由于Type 2 hypervisor意味着通过托管操作系统在分区间存在软件诱发的干扰通道，因此Type 2 hypervisor通常不被视为“强分区”。

值得注意的是，即便在系统处理器架构上正确实现Type 1 hypervisor以支持硬件虚拟化，若在硬件实施过程中有干扰，则强分区间仍存在干扰通道。例如，若各硬件虚拟化域间有共享缓存，或一个硬件虚拟化域发起的DMA可能会扰动另一硬件虚拟化域，但却未考虑hypervisor修复的可能性，则分区间仍存在潜在干扰。某些处理器体系结构考虑了此类干扰通道，比其他处理器体系更完整地实施了修复补救措施。因此，某些处理器体系结构比其他处理器体系结构更适合混合关键嵌入式系统。最后，即使存在此类基于硬件的干扰通道，type 1 hypervisor有时也可提供消除或最小化此类干扰通道的机制。对于以上举例，若关键虚拟机必须运行且不得受到基于硬件的干扰，则可暂停激活DMA的虚拟机或尽为（best-effort）虚拟机。尽管此类功能会为提高安全性和/或确定性而牺牲性能，但对于必须通过安全认证或满足严苛实时目标的系统中的Type 1 hypervisor，它们却是必备条件。

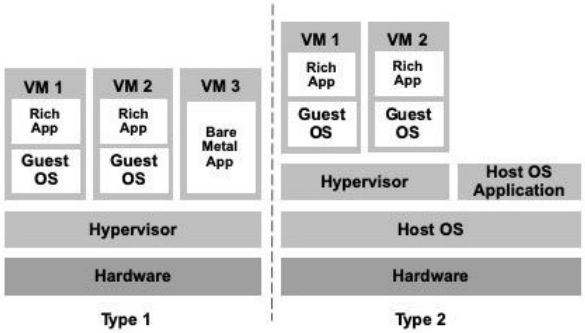


图2. Type 1与Type 2 Hypervisors对比

无论是Type 1还是Type 2 hypervisor，都有多种方法从虚拟机访问**物理设备**，包括直接访问hypervisor映射至虚拟机的设备特定的硬件寄存器。Hypervisor支持以下部分或全部功能：1）**直通设备**，其中设备被映射至虚拟机以直接访问和可选的设备中断； 2）**仿真设备**，其中物理设备接口以软件实现，因此可实现设备仲裁或设备分区等软件控制功能，以及 3）**虚拟设备**，其中可能存在也可能不存在用于实现虚拟驱动程序的物理设备。

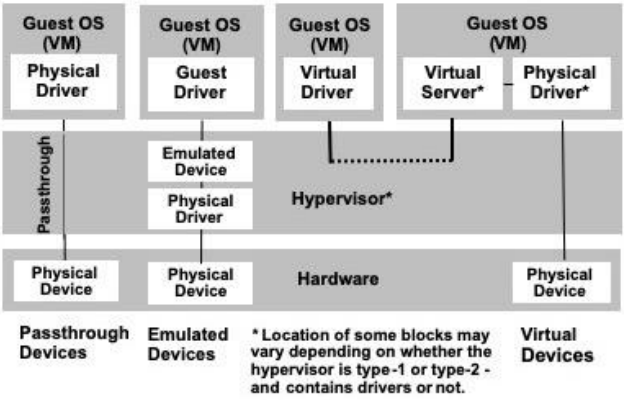


图3. 设备模型

某些 Type-1 hypervisor 提供最少的设备仿真和虚拟机调度，其主要重点是将硬件资源分区至虚拟机，强制执行这些分区，使用户基本上不知道Hypervisor的运行环境。这种主要专注于资源分区和分区执行的 Type-1 hypervisor有时被称为exokernels。安全可靠的多分区系统体系结构优化的一大潜力领域是将exokernel风格的hypervisor与裸机应用程序环境相结合，裸机应用程序环境包含单个地址空间和操作系统库（如RTOS或unikernel）。这种安排使各应用程序使用要求支持操作系统的功能，无需设置陷阱即可执行操作系统调用，从而实现以机器速度直接访问设备

IV. DISTRIBUTED SYSTEMS

When multicore-based systems are assembled from multiple partitions with each containing one or more applications and an OS instance, the overall runtime environment is a loosely-coupled distributed system running in a chip. Such systems require mechanisms for communications between the partitions so that applications running inside them can collaborate with each other. Although **shared memory** is possible and can be fast, shared memory must be used carefully so that it does not introduce interference channels or fault propagation paths between the partitions. Good system architectures reduce the size of failure domains and attack surfaces, not increase it. Shared memory-based communication mechanisms that are implemented with **unidirectional shared memory** (where a shared memory region is write-able by only a single virtual machine) are inherently safer than those that permit multiple writers since multiple writers can over-write each other erroneously or intentionally.

An alternative to used shared memory-based inter-partition communication is to leverage the hypervisor to move back safely between the virtual domains, using an emulated device or via a specialized **hypercall** interface. One reason to introduce hypercall-based inter-partition communication is for mixed-criticality systems so that the safety requirements on the safety related communication software implementation can be constrained to safety partitions and the certifiable hypervisor implementation. Meanwhile, the more flexible approaches can also be used by the best-effort non-certified operating systems such as the Linux or Windows instances which may be running in that same system.

With a safe inter-partition communication mechanism, resources that are managed in one partition can be leveraged by an application running in another partition. Examples of such resources are file systems, device drivers such as for real-time clocks or network interfaces, and OS services such as service name and log spooling services. This is true for partitions running on the same or separate hypervisors or directly on hardware in the same or different CPU cluster whether in the same multicore chip or across separate chips connected by a bus.

V. EXECUTION ISLANDS

Commercial multicore silicon and software environments have adopted the terminology "**Safety Island**" and "**Real-time Island**" to describe a hardware-partitioned or software-partitioned execution environment reserved respectively for safety-related or real-time related software. "**Security Island**" also has been used for a specialized security-related software hardware or software partition but is not as common.

A hardware-implemented safety or real-time island is typically a separate computer with memory, connectivity and processing cores – separate from the typically larger general-purpose compute silicon. These islands are able to operate as distinct execution environments without the support of the general purpose cores. It is sometimes the processor architecture of such islands is specialized from the general-purpose cores especially when the general-purpose cores are not suitable for such use due to share hardware that introduces interference channels.

Similarly, a software-implemented safety or real-time island is also a separate compute environment with dedicated memory, connectivity and processing cores. However, software-implemented safety or real-time islands are typically configured by a hypervisor from the general-purpose cores into distinct hardware partitions using hardware virtualization features of the processor. Safety and real-time islands are in practice useful only when configured by a Type 1 hypervisor because if a Type 2 hypervisor endeavoured to create them, then the safety or real-time requirements, including certification for safety islands or determinism for real-time islands, would fall to the underlying Type 2 hypervisor which they are typically not able to provide satisfactorily. This may change over time for mild levels of safety and/or real-time as, for example, Linux ™ has aspirations and some nascent levels of capability in these areas, e.g. the Linux Foundation's ELISA project.

VI. SUMMARY

Modern powerful multicore SoC hardware has so much capability that they have become integration platforms for software with varying core values including FOSS applications that are entangled with specific OS variants, software entangled with a security, safety or real-time OS, or bare metal apps. These applications require a software-based or hardware-based partitioning technology due to the multi-OS instances of the diverse applications being integrated. When there are safety or real-time requirements, this can be done with dedicated hardware islands or with a Type 1 hypervisor or a combination of the two. For cores sharing a memory bus, a hypervisor brings additional flexibility since it can be used to instantiate any number of general-purpose runtime partitions and also any of the island categories.

Such an architecture brings new levels of **architectural flexibility** so that design elements can be more easily updated and reused across product releases and product variants. **Operational flexibility** is also introduced since the lifecycle of partitions are distinct from each other, which enables **continuous integration** and **continuous deployment** and also dynamic **horizontal and vertical scaling** due to the loose coupling and distribution systems architecture.

Although introducing a hypervisor into the application + OS runtime stack does introduce a third operational layer, having a separate layer to focus on resource partitioning and partition enforcement does bring focus to the role of each layer and enable general purpose compute, safety and real-time to be separated out so that each type of runtime can be optimized for what it does best. Although the hypervisor would appear to introduce interrupt latency to systems, modern type 1 virtualization hardware enables the implementation of **direct interrupts** which are delivered directly to the guest and thus no additional interrupt latency is introduced when a hypervisor is in the system architecture. In addition, developers can benefit from the hypervisor level since with the **debug features** of a hypervisor, they are able to debug OS partitions and bare metal apps remotely.

In particular, it is important to acknowledge that with the increased amount of software in complex applications coming in large part from the FOSS community and such software now comes with its own OS assumptions, the model of porting such

software to a unifying OS such as an SMP OS or a traditional microkernel that runs on all cores is becoming less common now that hypervisors with hardware support can enable such software to run in its native environment while enabling real-time and optionally safety at the time well. Such an architecture is expected by the author to save system architects time and embedded systems companies money as they are able to enjoy the architectural and operational flexibility of easily running software with diverse core values on the same hardware platform – and by leveraging previously sunk non-recurring engineering costs through increased re-use of stable software partitions.

Based on the above points, the following are the attributes to consider when selecting a Type 1 hypervisor for a mixed criticality system:

1. Hard real-time guest support (i.e. it must be hard real-time itself).
2. Availability of safety certification criteria for the hypervisor
3. Device models supported for drivers in bare metal apps or guest OSs.
4. Resource sharing between guests and CPU clusters.
5. Strong partitioning that minimizes HW interference channels.
6. Scalability to a larger number of cores (i.e the hypervisor is implemented using lock-free techniques and avoids order N algorithms).
7. Direct interrupts so interrupts can bypass the hypervisor and go directly to a VM to enable native-speed interrupt processing performance.
8. Silicon independence of (portability to) CPU SKUs and processor architectures.
9. Support for both UP and SMP guest OSs.
10. Support for fractional core scheduling (running than one VM on a core)
11. Support for VM preemption (e.g. RTOS preempts best effort OS such as Linux on the same core)
12. Support for unmodified guest OSs such as Android, Linux, Windows, etc.
13. Support for individual device access (e.g. PCI devices mapped to VMs on a per VM basis)
14. Enablement of a rich development and tooling experience via hypervisor debug features