Virtualisierung im industriellen Umfeld
Grundlagen, Lösungen, Erfahrungen

Frank Erdrich und Jan von Wiarda, emtrion

Virtualisierung bietet eine Lösung, um mehrere Softwareinstanzen, etwa Betriebssysteme, auf einem System zu betreiben. Dabei laufen die einzelnen Systeme getrennt voneinander ab ohne zu wissen, dass andere Instanzen auf dem selben physikalischen System ausgeführt werden. Durch diese Trennung kann eine gewisse Sicherheit erreicht werden, da beispielsweise eine komprommitierte Instanz nicht auf die Ressourcen, etwa den Hauptspeicher, anderer Instanzen zugreifen kann.

Im industriellen Umfeld mit eingebetteten Systemen wurde eine solche Trennung in der Regel durch einzelne Rechensysteme mit dedizierten Kommunikationskanälen aufgebaut. Dazu sind beispielsweise zwei Prozessoren auf einer Steuereinheit verbaut, die über ein definiertes Protokoll etwa über Ethernet oder SPI miteinander kommunzieren. Damit kann eines der der Systeme die Kommunikation zur Außenwelt etwa in Form eines MMI (Mensch-Maschine Interface) abdecken, während auf dem zweiten System eine sicherheitskritische Software ausgeführt wird, welches Echtzeitbedingungen einhalten muss.

Durch die Entwicklung immer leistungsfähigeren Prozessoren mit mehreren Kernen, die hinsichtlich der Leistungsaufnahme und damit der Abwärmeerzeugung auch für eingebettete Systeme geeignet sind, ist die Virtualisierung zur Option für industrielle Geräte geworden, um teure Multi-Prozessorlösungen durch günstigere Single-prozessorsysteme, dafür mit mehreren Kernen, zu ersetzen. Dennoch sind einige Voraussetzungen zu erfüllen, damit ein Prozessor in Virtualisierungsprojekten einsetzbar ist. Nachfolgend soll nun beschrieben werden, was für eine Virtualisierung von Systemen notwendig ist, welche Arten von Virtualisierung existieren und wie eine Lösung für ein eingebettetes System aussehen kann. Dabei wird speziell auf die im Umfeld der eingebetteten Systemen weit verbreiteten SoCs (System on Chips) mit ARM-Kern eingegangen.

Virtualisierungstypen
Virtualisierung kann in verschiedene Arten aufgeteilt werden, die jeweils andere Ansätze verfolgen und verschiedene Anforderungen an die Unterstützung im Prozessor stellen. Vorgestellt werden hier nur diejenigen Typen der Virtualisierung, die auf einem eingebetteten Systen sinnvoll zu betreiben sind. Diese Betrachtung kann sich durch erscheinen weiterer, noch leistungsfähiger Prozessoren verschieben.

Die hierbei wichtigen Virtualisierungsarten sind Partitionierung sowie die Virtualisierung mittels eines Typ-1 oder Typ-2 Hypervisor. Virtualisierung durch Emulation (Qemu) und Anwendungsvirtualisierung (Java VM) werden hier nicht weiter betrachtet.

Partitionierung
Mittels Partitionierung werden die im System verfügbaren Ressourcen einzelnen Instanzen zugewiesen. Eine Instanz kann nur die Ressourcen nutzen, die ihr zugewiesen wurden. Die Zuweisung der Ressourcen wird durch eine

Verwaltungsinstanz vorgenommen und kann beispielsweise durch eine beschreibende Konfigurationsdatei vom Benutzer erstellt und angepasst werden.

Typ-2 Hypervisor

Unter einem Typ-2 Hypervisor wird ein System verstanden, welches vollständig virtualisiert wird. Dabei wird ein vollständiges virtuelles System erstellt, welches beispielsweise auch das BIOS auf x86-basierten Plattformen umfasst. Die Ausführung innerhalb einer virtuellen Maschine wird dabei nicht emuliert, wie etwa bei Qemu, sondern auf der Hardware selbst ausgeführt. Das Besondere an einem Typ-2-Hypervisor ist die Tatsache, dass dieser dem Gastbetriebssystem virtuelle Ressourcen wie virtuelle CPUs, Hauptspeicher, Festplatten, Netzwerkkarten etc. zur Verfügung stellt. Das unterscheidet ihn von Typ-1-Hypervisorer. Ein Hypervisor wird auch als Virtual Machine Monitor bezeichnet.

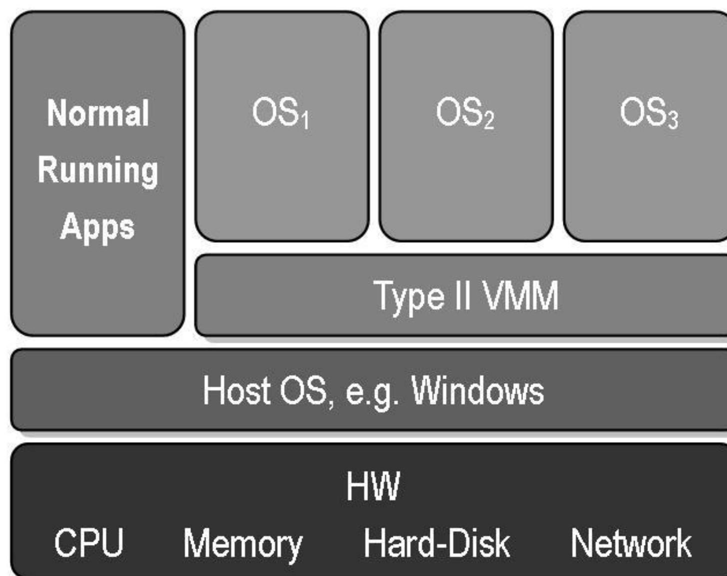

Abbildung 1: Typ-2 Hypervisor (Quelle: Wikipedia - public domain)

Des Weiteren wird ein Typ-2-Hypervisor als Anwendung innerhalb des Host-Betriebssystems ausführt. Er verteilt die physikalischen Hardwareressourcen des Rechners an die Gastsysteme. Es kann dabei vorkommen, dass der Hypervisor als Emulator dient und zwar dann, wenn auf bestimmte Hardware nicht gleichzeitig zugegriffen werden kann, wie beispielsweise eine physikalische Netzwerkkarte. Ein Vorteil der Emulation ist die Vermeidung von Treiberproblemen, hat jedoch einen gewissen Performanceverlust zur Folge.

Das Prinzip der vollständigen Virtualisierung basiert auf unterschiedlichen Privilegierungsebenen eines Prozessors und ist architekturabhängig. Die unterschiedlichen Ebenen unterscheiden sich durch den Umfang an Zugriffsrechten auf CPU Register, Hauptspeicherbereiche und Hardware. Der Hypervisor befindet sich in der Ebene mit den höchsten Rechten. Die Gastbetriebsysteme laufen in niedrigeren Privilegierungsebenen. Wenn ein Gast versucht, höherprivilegierte Betriebssystemoperationen durchzuführen, fängt der Hypervisor diese als

sogenannten Exceptions ab, wertet sie aus und behandelt sie entsprechend. Nur über den Hypervisor kann ein Gast auf die Hardware und geteilte Systemressourcen zugreifen. Bekannte Vertreter von Softwarelösungen im Bereich der vollständigen Virtualisierung sindbeispielsweise Vmware Workstation, KVM und VirtualBox.

Der Vorteil der vollständigen Virtualisierung ist, dass keine Änderungen am Host- und Gastbetriebssystem erforderlich sind. Den Zugriff auf Systemressourcen wird vom Hypervisor abgefangen, verwaltet und and das Host-System durchgereicht. Dies macht die Ausführung des Gastes im Vergleich zur nativen Ausführung recht performant. Die Wechsel der Privilegierungsebenen führen zu einem gewissen Performanceverlust, da jeder Wechsel der Ebene, ähnlich wie bei Taskwechseln in einem Echtzeitbetriebssystem, Kontextwechsel zur Folge hat. Ein Kontextwechsel führt dazu, dass verschiedene Register der CPU gesichert und wiederhergestellt werden müssen. Wenn ein privilegierter Befehl vom Gast angefordert wird, muss der Hypervisor diesem eine Wrapperfunktion zur Verfügung stellen, welche die eigentliche Ausführung an die Kernel-API des Hosts weiterleitet.

Typ-1 Hypervisor

Im Gegensatz zur vollständigen Virtualisierung der Typ-2 Hypervisor wird bei der Paravirtualisierung durch Typ-1 Hypervisor keine Hardware virtualisiert oder emuliert. Stattdessen läuft der Typ-1 Hypervisor direkt auf der Hardware auf der höchsten Privilegierungsebene. Aus diesem Grund wird diese Art von Hypervisor auch als bare-metal Hypervisor bezeichnet. Gast-Betriebssysteme greifen nicht auf emulierte Hardware, sondern auf eine vom Hypervisor bereitgestellte API zu. Physikalische Ressourcen werden einzig und allein über diese API angesprochen.

Der Hypervisor fungiert hier als eine Art Betriebssystem, dessen Aufgabe es ist, die Systemressourcen unter den Gästen zu verteilen. Dies ermöglicht die getrennte Ausführung von Betriebssystemen auf der gleichen Hardware. Im Falle des Xen Hypervisors läuft zusätzlich ein Host-Betriebssystem auf der zweit-höchsten Privilegierungsebene, dessen Kernel dadurch keine Operationen ausführen kann, die die höchste Privilegierungsebene erfordern. Zu diesem Zweck kommuniziert der Host-Kernel über sogenannte Hypercalls mit dem Hypervisor. Dies ist vergleichbar mit der Kommunikation über System Calls zwischen dem Kernel- und dem Userspace bei Betriebssystemen.

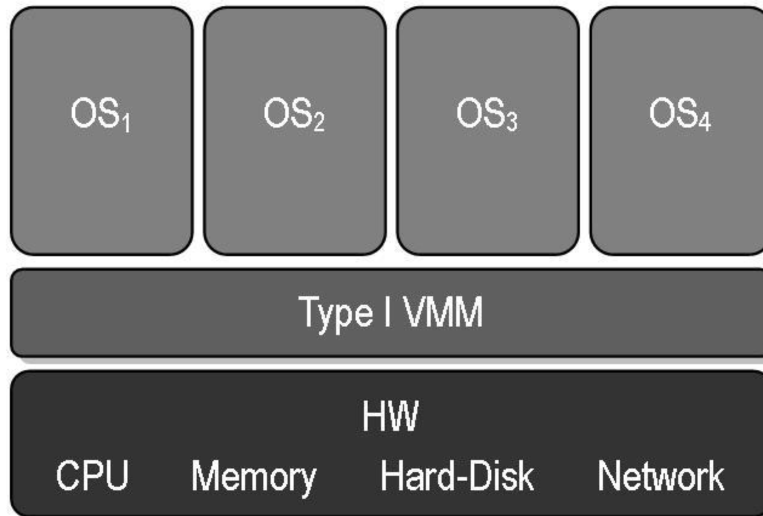

Abbildung 2: Typ-1 Hypervisor (Quelle: Wikipedia - public domain)

Ein Typ-1 Hypevisor erfordert eine Anpassung des Host-Kernels, damit dieser die genannten Hypercalls unterstützt. Der Hypervisor fängt somit alle System Calls der Gäste ab und leitet diese an den Host-Kernel weiter. Dieser muss den jeweiligen System Call in einen Hypercall umwandeln und diesen wiederum an den Hypervisor weiterleiten, der dann die eigentliche Operation auf der Hardware ausführt. Bekannte Typ-1-Hypervisor-Systeme sind unter anderem VMware ESXi und Xen.

Virtualisierung auf ARM

Die 32-Bit Architektur ARMv7 zum aktuellen Zeitpunkt die am weitesten verbreitete Architektur im Bereich von Embedded Linux. Parallel dazu existiert die 64-Bit Architektur ARMv8, deren Verbreitung durch aktuelle SoCs verschiedener Hersteller steigt. Trotz allem ist nicht jeder SoC mit ARM-kern für Virtualisierung geeignet. Um virtualisierung zu unterstützen, müssen die ARM Virtualization Extensions und die ARM Security Extensions verfügbar sein. Das sind beispielsweise Kerne vom Typ Cortex-A7, Cortex-A15 oder Cortex-A53. Im nachfolgenden Abschnitt wird versucht, einen grundlegenden Überblick über die Virtualisierung auf ARM-Kernen sowie den damit verbundenen Subsystemen des Kerns aufzuzeigen. Die Informationen sind Aufgrund des Umfangs und der Komplexität nur rudimentär und teilweise unvollständig dargestellt und sind bevorzugt mit der Dokumentation von ARM zu lesen (*ARM Architecture Reference Manual).*

ARM Security Extensions (TrustZone)

Die Security Extensions führen einen neuen CPU Mode ein, den Monitor Mode. Zusätzlich werden der Architektur zwei Security States, der Non-Secure State und der Secure State, hinzugefügt. Diese beiden States werden Normal World und Secure World genannt. Die CPU befindet sich stets in einem der beiden Zustände. Über die neue Instruktion SMC oder durch eine Hardware Exception (IRQ, FIQ, etc.) wird in den Secure Monitor Mode gewechselt. Dieser Mode wird später vom Hypervisor dazu verwendet um die CPUs nach dessen Ausschaltung per PSCI wieder freizugeben. Eine Koexistenz zwischen Hypervisor und der Secure World ist ohne Probleme möglich. Die Security Extensions bieten unter anderem erweiterte

Hardware Security Features an, die die Implementierung von DRM, Secure Payment und anderen Software-Lösungen ermöglichen. Eine Bekannte Lösung für TrustZone ist beispielsweise OP-TEE. (https://wiki.linaro.org/WorkingGroups/Security/OP-TEE)

ARM Virtualization Extensions
Die Virtualization Extensions führen ebenfalls einen neuen CPU Mode ein, den sogenannten Hyp Mode. In diesem Modus wird der Hypervisor ausgeführt. Darüber hinaus werden zahlreiche zusätzliche Register und Hardwarefunktionen wie die Erweiterung der MMU, des Interrupt Controllers (GIC) und weiteren Bausteinen um bestimmte Virtualisierungsfunktionen, eingeführt. Insbesondere wird die Virtualisierung des Non-Secure States der ARM VMSAv7 Implementierung unterstützt. Zusätzlich wird die neue Instruktion HVC eingeführt, mit der virtuelle Maschinen mit dem Hypervisor kommunizieren können. Sobald eine Architektur die Virtualization Extensions unterstützen will, müssen zwangsweise die Security Extensions, die Large Physical Address Extension (LPAE) und die Multiprocessing Extensions implementiert sein. Ein virtualisiertes System umfasst nach ARM Definition im wesentlichen:

- einen Hypervisor, der im Non-Secure PL2 Hyp Mode ausgeführt wird und für das Umschalten zwischen den Gast-Betriebssystemen verantwortlich ist
- eine gewisse Anzahl an Gast-Betriebssystemen, die jeweils im Non-Secure PL1 und PL0 Modus ausgeführt werden
- für jedes Gast-Betriebssystem Anwendungen, die üblicherweise im User Mode laufen

Entscheidender Vorteil der Virtualization Extensions ist die Einführung von Hardware-Mechanismen, die verhindern, dass der Hypervisor per Software während der Ausführung von Gast-Betriebssystemen zu oft eingreifen muss. Dies würde die Performance massiv in negativem Maße beeinflussen. Dank dieser Hardware Mechanismen muss der Hypervisor sich um folgende Aufgaben nicht kümmern, sondern kann dies den Gast-Betriebssystemen überlassen:

- Interrupt Masking
- Page Table Management
- Gerätetreiber durch Hypervisor Memory Relocation
- Teile der Kommunikation der Gäste mit dem Interrupt Controller (GIC)

Speicher, IRQ und I/O
Anders als bei der klassischen Übersetzung von virtuellen Speicheradressen in physikalische Speicheradressen mithilfe von Seitentabellen (page tables), finden im Virtualisierungsumfeld nicht nur eine sondern mehrere Übersetzungsphasen statt. Die VE bieten sogenannte translation regimes für Speicherzugriffe von den Non-Secure PL0 und PL1 Modi in den Non-Secure PL2 Modus an. Im Non-Secure PL0 und PL1 translation regime geschieht die Adressübersetzung in zwei Phasen, stages genannt. Stage 1 mapped die Virtual Address (VA) in eine Intermediate Physical Address (IPA). Typischerweise konfiguriert und kontrolliert die Gast-VM diese Phase und glaubt, dass die IPA die Physical Address (PA) ist. Stage 2 mapped dann die IPA zu einer PA. Diese Phase kontrolliert der Hypervisor mit Unterstützung der MMU transparent für die Gast-VM.

Auftretende Interrupts hingegen sind erst einmal nicht einem konkreten Empfänger zuzuordnen. Ein solcher Empfänger kann die gerade ausgeführte VM sein, eine gerade nicht ausgeführte VM oder aber der Hypervisor selbst. Der Ansatz der VE sieht vor, dass physikalische Interrupts zunächst an den Hypervisor geschickt werden. Falls der Interrupt zu einer VM gehört, mapped der Hypervisor einen virtuellen Interrupt in diese VM. Um diese Funktionalität zu unterstützen, bedarf es der Erweiterung des Interrupt Controllers. Genauer bieten die VE spezielle Register, um die virtuellen Interrupts zu verwalten. Es gibt physikalische und virtuelle Register. Der Hypervisor greift auf die physikalischen Register zu, die VMs auf die virtuellen. Die virtuellen Register werden vom Hypervisor geremapped, so dass die VM denkt sie würde auf die physikalischen Register zugreifen. Die Interrupt Mask Bits I, A und F im CPSR gehören ausschließlich der jeweils gerade ausgeführten VM und müssen daher nicht mehr getrapped werden. Virtuelle Interrupts werden direkt in den jeweiligen CPU Mode (FIQ, IRQ, Abort) weitergeleitet. Die VM manipuliert den virtualisierten Interrupt Controller. Dazu wurde eine virtuelle GIC Schnittstelle eingeführt. Die Interrupt Service Routine (ISR) der VM kommuniziert dabei mit dem virtuellen GIC. Es gibt Listen, die Pending und Active Interrupts für jede VM enthalten. Der virtuelle GIC kommuniziert dabei mit dem physikalischen GIC. Die Funktionalität der VM ändert sich nicht durch den virtuellen GIC. Der Hypervisor kann IRQs im HCR Register als virtuell setzen. Interrupts werden so konfiguriert, dass diese eine Hypervisor Trap auslösen. Der Hypervisor kann ein Interrupt über sogenannte register lists an eine CPU weiterleiten, die gerade der VM zugeordnet ist.

ARMv7 nutzt zur Kommunikation mit Peripheriegeräten Memory-mapped I/O, das heißt die CPU kommuniziert über in Hauptspeicher gemappte Geräteregister mit den Geräten. Zwischen CPU und I/O-Geräten sitzt die MMU, die die Zugriffe regelt. Bei den Adressen der Geräteregister handelt es sich um physikalische Speicheradressen, das heißt die MMU muss keine Übersetzung vornehmen sondern gibt die Adressen direkt weiter. Nur der Zugriffsschutz ist von Bedeutung, da hier auch der Hypervisor ansetzt um die Gäste voneinander zu trennen.

Der Jailhouse Hypervisor

Das Softwareprojekt Jailhouse steht unter der Leitung der Siemens AG und ist unter den Lizenzbedingungen der GPLv2 als Open Source Projekt realisiert. Der Quellcode ist unter GitHub hinterlegt. Es gibt einen Master Branch und einen Next Branch. An der Entwicklung ist neben Siemens auch ARM und Huawei beteiligt. Jailhouse wurde 2013 ins Leben gerufen und unterstützt neben ARM auch die x86 und x86-64 Architekturen.

Konzept

Jailhouse ist ein sogenannter partitionierender Hypervisor. Er wird dazu benutzt, sogenannte asymmetrische Multiprozessorsysteme (AMP) auf Linux Basis zu realisieren. Das bedeutet, neben einem Standard Linux Kernel können auf der gleichen Hardware, dem gleichen SoC, verschiedene andere Betriebssysteme oder Bare-metal Anwendungen ausgeführt werden, die soweit möglich strikt voneinander getrennt werden. Der Jailhouse Hypervisor sorgt dabei für die Isolation zwischen den sogenannten Zellen und bedient sich hierbei der Hardwareunterstützung von modernen CPU-Kernen. In jeder Zelle läuft ein Gast, der über bestimmte Systemressourcen wie zum Beispiel CPUs, Hauptspeicherbereiche, und I/O-Geräte verfügt, über die er volle Kontrolle hat. Der Hypervisor sorgt dabei nur für die strikte

Trennung der Zellen. Das Hostsystem, das auf einem ganz normalen Linux Kernel basiert, wird dabei als Root Zelle bezeichnet. Alle zusätzlich ausgeführten Systeme werden Non-root Zellen genannt. Die Root Zelle dient dazu, den Hypervisor und weitere Zellen zu starten und zu kontrollieren. Sie verfügt jedoch nicht über die komplette Kontrolle aller Systemressourcen, sondern ihr werden beim Erzeugen weiterer Zellen Ressourcen wie CPUs oder Speicher entzogen. Jailhouse emuliert keine Ressourcen, die nicht vorhanden sind, sondern teilt lediglich die vorhandene Hardware auf die einzelnen Zellen auf. Daher findet auch kein Scheduling zwischen den einzelnen Gästen statt. Die innerhalb einer Non-root Zelle ausgeführte Software, sei es nun ein Linux Kernel, FreeRTOS oder eine Bare-metal Anwendung, wird von den Entwicklern als Inmate bezeichnet.

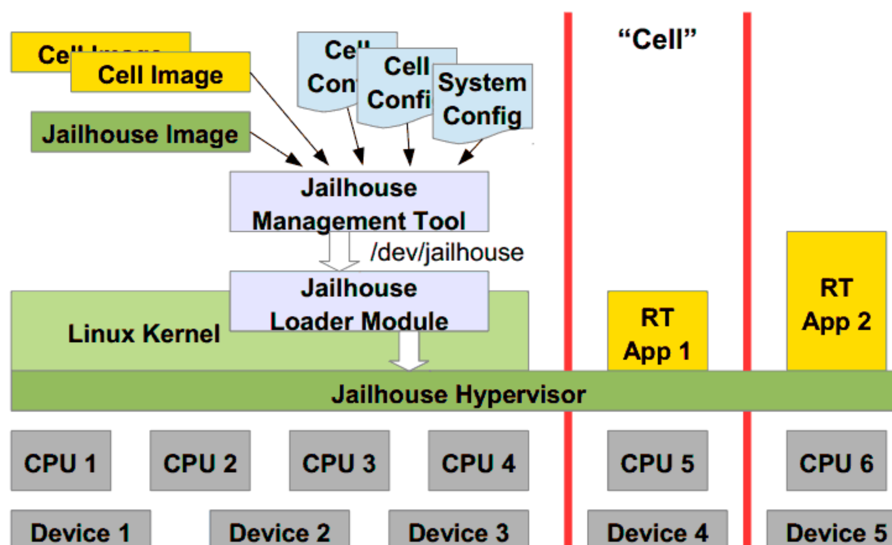

Abbildung 3: Jailhouse Hypervisor Architektur (Quelle: Siemens)

Motivation

Eine wesentliche Motivation für die Entwicklung eines partitionierenden Hypervisors ist der Bedarf an Virtualisierungslösungen, mit denen Echtzeitfähigkeit ermöglicht werden soll. Echtzeitfähigkeit sowie Safety und Security sollen die Besonderheit von Jailhouse sein. Dazu zählt auch das Ziel, eine möglichst kleine Codebase zu erhalten, um zum Einen die Komplexität möglichst gering zu halten um ein Verständnis des Gesamtsystems zu ermöglichen und zum Anderen die Codequalität auf einem hohen Niveau zuhalten. Der eigentliche Hypervisor umfasst nur knapp 12.600 LOCs, wohingegen Xen mit knapp 200.000 LOCs schon erheblich komplexer ist. Natürlich kann Jailhouse nicht mit dem Funktionsumfang von Xen gleichgesetzt werden, dies ist aber auch nicht die Intension hinter dem Projekt. Ein Augenmerk liegt auf der Codequalität. Seit Beginn der Entwicklung wurde regelmäßig mithilfe des statischen Code-Analyse Tools Coverity Scan die Codebasis auf Fehler überprüft. In der letzten Version des Quellcodes wurde von Coverity eine Fehlerdichte von 0.00 ermittelt. Dies bedeutet natürlich keinesfalls, dass die Software fehlerfrei ist, deutet jedoch auf eine sehr gute Codequalität hin. Zum Vergleich wurde bei Xen eine Fehlerdichte von 0.56 ermittelt. Die Besonderheit des Hypervisors ist zudem die Tatsache, dass es nicht wie bei Xen vom Bootloader als BLOB geladen werden muss oder wie bei KVM den Linux Kernel selbst zu einem

Hypervisor macht, sondern innerhalb eines unveränderten Linux Systems geladen und konfiguriert werden kann. Der Fokus von Jailhouse liegt dabei nicht in der Ablösung von KVM oder Xen im Desktop Bereich, sondern in der Realisierung von Systemen, die kritische Anwendungen im Bereich „Funktionale Sicherheit" umfassen. Als Beispielanwendung könnte ein System realisiert werden, in dem eine GUI unter Qt in einer Zelle läuft und auf dem gleichen SoC innerhalb einer anderen Zelle eine zeit- und sicherheitskritische Maschinensteuerung ausgeführt wird. Diese dürfen sich selbstverständlich unter keinen Umständen in ihrer Ausführung beeinflussen. Dafür soll der Jailhouse Hypervisor sorgen.

Systemvoraussetzungen
Voraussetzung für den Einsatz von Jailhouse auf ARM basierten Systemen ist das Vorhandensein der ARMv7 Virtualization Extensions. Darüber hinaus müssen auf dem SoC
mindestens zwei CPU-Kerne vorhanden sein, da wie erwähnt eine Partitionierung stattfindet und somit pro Zelle mindestens ein CPU-Kern zugewiesen werden muss. Der Bootloader, in unserem Falle U-Boot, muss außerdem Unterstützung für den Einsatz in Virtualisierungsumgebungen bieten, das heißt er muss Linux im sogenannten HYP-
Mode starten. Zusätzlich muss die PSCI Unterstützung vorhanden sein, die das Offline Schalten von CPUs ermöglicht.

Laden und Aktivieren des Hypervisors
Der Hypervisor selbst ist ein etwa 30 Kilobyte großes BLOB. An der geringen Größe des Hypervisors kann man die oben erwähnte Absicht einer schmalen Codebasis erken-
nen. Dieses BLOB kann mithilfe des Treibermoduls jailhouse.ko über die Kernel-API als Firmware in einen dedizierten RAM Bereich kopiert werden, der beim Booten des
Linux Systems reserviert werden muss. Über die Kernelparameter mem=958M vmalloc=512M wird dem Kernel mitgeteilt, dass nicht der volle, 1 GiB große Adress-
raum genutzt werden, sondern im oberen Bereich Platz für den Hypervisor reserviert werden soll. Das Treibermodul erzeugt zudem noch einen Deviceknoten /dev/jailhouse, der als Schnittstelle für das Userspace Programm /usr/local/sbin/jailhouse dient. Der Treiber selbst enthält aber keinerlei Hypervisor Logik sondern dient lediglich dem Laden des Hypervisors. Über das Userspace Programm können verschiedene Aktionen ausgeführt werden, wie beispielsweise das Starten und Stoppen einer Non-Root Zelle.

Funktionsweise des Hypervisors
Sobald der Treiber über modprobe jailhouse geladen und der Hypervisor über die Userspace Anwendung aktiviert wurde, setzt dieser sich zwischen die eigentliche Hardware und die Root Zelle. Bei der binären Datei bananapi.cell handelt es sich um eine in C geschriebene Konfigurationsdatei, die aus mehreren verschachtelten Structs besteht und die jeweilige Hardware beschreibt. Dabei wird festgelegt an welcher physikalischen Adresse der Hypervisor ausgeführt wird, wie groß dessen Speicherbereich ist und wie viele CPUs die Hardware umfasst. Zusätzlich wird die Memory Mapped I/O-Adresse einer UART Debug Console eingetragen, über die der Hypervisor Ausgaben loggt. Außerdem werden sämtliche Memory Mapped I/O-

Regionen der Peripheriegeräte wie zum Beispiel SPI, MMC, USB, SATA, etc. eingetragen. Die Register der jeweiligen Controller sind in diese Bereiche gemappt und der Hypervisor muss dies natürlich wissen um Zugriffe auf die Speicherbereiche überwachen zu können. Der Hypervisor wird immer im Non-Secure PL2 Hyp Mode ausgeführt.

In diesem CPU Modus hat der Hypervisor Zugang zu wichtigen Registern, die der Verwaltung dienen. Auch nachdem der Hypervisor aktiviert wurde behält die Root Zelle zunächst alle CPUs zur eigenen Verfügung, bis weitere Zellen gestartet werden. Bereits jetzt überwacht der Hypervisor hardwareunterstützt jeden Speicherzugriff und nimmt sämtliche Interrupt-Requests der Hardware entgegen, um diese an die Root Zelle weiterzuleiten. Auch beim Interrupt-Handling bedient sich der Hypervisor der Hardwareunterstützung des GICv2 Interrupt-Controllers von ARM. Falls nun während der Ausführung einer Root oder Non-Root Zelle eine sogenannte Exception eintritt, wird vom Hypervisor abhängig vom jeweiligen CPU-Mode ein bestimmter Exception-Handler aufgerufen, der die Ausnahmebehandlung übernimmt. Ab diesem Zeitpunkt übernimmt der Hypervisor die CPU. Die Exception-Handler werden über feste Vektoradressen, die in dersogenannten hyp_vectors Tabelle stehen, angesprungen.

Je nach Art der Ausnahme, werden durch den Hypervisor entsprechende Maßnahmen durchgeführt. Greift eine Zelle zum Beispiel auf eine Speicheradresse zu, für die sie keine Berechtigung hat, wird die Funktion arch_dump_exit() aufgerufen, die letztlich die CPU der betroffenen Zelle parkt. Das bedeutet die Zelle ist nicht mehr operabel. Die anderen Zellen sind davon nicht betroffen und werden weiterhin ausgeführt. Bei einem unerlaubten Speicherzugriff spricht man von einem Abort, der von der MMU generiert wird, also wiederrum durch Hardwareunterstützung.

Des Weiteren unterscheidet man bei Jailhouse zwischen Exceptions und Traps. Unter Exceptions versteht man Ausnahmebehandlungen, wie sie bei Interrupts oder undefinierten Instruktionen auftreten. Im Jailhouse Kontext existieren genau genommen nur zwei richtige Exception Handler, je einer für physikalische Interrupts und für Software generierten Interrupts. Alle anderen Ausnahmen werden von Trap Handlern bearbeitet, die von Exception Handlern aufgerufen werden. Traps können durch Zugriffe auf Co-Prozessor Register auftreten. Hypervisor Calls (HVC) können von Zellen getriggert werden um mit dem Hypervisorzu kommunizieren, beziehungsweise um beim Hypervisor bestimmte Funktionalitäten anzufordern. Kommt es in Zellen zu HVC Instruktionen, werden diese von der Funktion arch_handle_hvc() behandelt. Secure Monitor Calls (SMC) werden ebenfalls vom Hypervisor getrappt. Dabei handelt es sich um eine Instruktion, die in CPUs mit den Security Extensions auftreten kann. Diese ARM CPUs können sich, wie oben beschrieben, in zwei verschiedenen Security States befinden, dem Secure State und dem Non-Secure State. Mithilfe der SMC Instruktion kann der Security State geändert werden. Diesen Fall fängt der Hypervisor ab. Eine weitere Trap wird wie oben beschrieben bei unerlaubtem Speicherzugriff (Data Abort) ausgelöst.

Erfolgreich getestete Inmates

Als erstes Inmate wurde eine FreeRTOS Anwendung ausgeführt, nachdem zusammen mit dem Entwickler eine Anpassung einer halbwegs fertigen Lösung 2 auf GitHub stattgefunden hatte. Aufbauend auf dieser Lösung wurde eine Bare-Metal

Anwendung als Inmate zum Laufen gebracht, die periodisch die LED an- und ausgeschaltet hat.

Das zweite Inmate sollte ein Linux Kernel sein, was nur mit einigem Aufwand gelang. Zunächst musste herausgefunden werden, wo und wie das Kernel zImage in den Speicher geladen werden muss und wie es danach ohne Bootloader ausgeführt werden kann. Dabei half das Studium von U-Boot Code und des ARM Bootprotokolls in der offiziellen Kerneldokumentation.

Geteilte Ressourcen und Benchmarks
Eine in Hinblick auf Echtzeitfähigkeit berechtigte Frage ist die nach den geteilten Systemressourcen eines SoCs und inwiefern die Ausführung von Zelle A eine andere Zelle B beeinflusst. Wenn man die CPUs betrachtet, betrifft dies in erster Linie die externen Schnittstellen an Busse und Caches. Speziell die Bus Interface Unit (BIU), das Level 2 Cache System samt Snoop Control Unit (SCU) und Generic Interrupt Controller (GIC) sowie die Anbindung an den AMBA 4 Bus über die AXI Coherency Extensions (ACE) werden zwischen den CPU-Kernen geteilt. Dies zeigt, dass auf aktueller Hardware eine Beeinflussung verschiedener Zellen untereinander natürlich nicht komplett vermieden werden kann, selbst mit dem Partitionsansatz von Jailhouse.
Dank der Partitionierung verhindert man jedoch das CPU-Scheduling und die damit verbundenen Kontextwechsel. Zudem haben Benchmark Tests von Seiten des Jailhouse Projekts ergeben, dass die Flaschenhälse, die ohnehin nicht vermieden werden können, keinen allzu großen Einfluss auf die Echtzeitfähigkeit haben. Es gibt jedenfalls keinen effizienteren Ansatz mit der vorhandenen Hardware.

Um verschiedene Systeme im Hinblick auf den Overhead des Hypervisors zu vergleichen, wurde ein auf der Anwendung cpuburn basierende Benchmark erstellt. Dabei wurden Xen und Jailhouse untersucht. Hierbei konnte nachgewiesen werden, dass der Overhead von Jailhouse minimal ist und die Hardware ideal ausnutzt. Im Vergleich zu einem Linux Kernel ohne Hypervisor belief sich der Overhead bei Jailhouse auf 0,04 %, bei Xen hingegen auf 5,71 %. Dies ist vermutlich in erster Linie auf das CPU-Scheduling von Xen zurückzuführen.

Fazit
Mit Jailhouse liefert Jan Kiszka von Siemens eine stabile und schnelle, echtzeitfähige Möglichkeit, einen Multikern-SoC derart zu partitionieren, dass mehrere Gastsysteme parallel ausgeführt werden können, ohne sich gegenseitig zu beeinflussen. Damit besteht eine Variante, Securityanwendungen parallel mit einem unsicheren Softwareteil gemeinsam auf einem SoC auszuführen bzw. zu virtualisieren und damit den Hardwareaufwand und die damit verbundenen Kosten zu reduzieren. Verständlicherweise muss für jedes Projekt gezielt überprüft werden, ob die durch die Virtualisierung vorhandenen, wenn auch geringen, Performanceeinbußen die Echtzeitfähigkeit nicht beinflussen.