How virtualization makes power management different

Kevin Tian, Ke Yu, Jun Nakajima, and Winston Wang
Intel Open Source Technology Center
{kevin.tian,ke.yu,jun.nakajima,winston.l.wang}@intel.com

Abstract

Unlike when running a native OS, power management activity has different types in a virtualization world: virtual and real. A virtual activity is limited to a virtual machine and has no effect on real power. For example, virtual S3 sleep only puts the virtual machine into sleep state, while other virtual machines may still work. On the other hand, a real activity operates on the physical hardware and saves real power. Since the virtual activity is well controlled by the guest OS, the remaining problem is how to determine the real activity according to the virtual activity. There are several approaches for this problem.

1. Purely based on the virtual activity. Virtual Sx state support is a good example. Real S3 sleep will be executed if and only if all the virtual S3 are executed.
2. Purely based on the global information, regardless of the virtual activity. For example, CPU Px state can be determined by the global CPU utilization.
3. Combination of (1) and (2): in some environments, VM can directly access physical hardware with assists from hardware (e.g., Intel Virtualization Technology for Directed I/O, a.k.a. VT-d). In this case, the combination of (1) and (2) will be better.

This paper first presents the overview of power management in virtualization. Then it describes how each power management state (Sx/Cx/Px) can be handled in a virtualization environment by utilizing the above approaches. Finally, the paper reviews the current status and future work.

1 Overview of Power Management in Virtualization

This section introduces the ACPI power management state and virtualization mode, and later the overview of a power management implementation in virtualization.

1.1 Power Management state in ACPI

ACPI [1] is an open industry specification on power management and is well supported in Linux 2.6, so this paper focuses on the power management states defined in the ACPI specification.

ACPI defines several kinds of power management state:

- **Global System state (G-state)**: they are: *G0* (working), *G1* (sleeping), *G2* (soft-off) and *G3* (mechanical-off).
- **Processor Power state (C-state)**: in the *G0* state, the CPU has several sub-states, $C0 \sim Cn$. The CPU is working in *C0*, and stops working in $C1 \sim Cn$. $C1 \sim Cx$ differs in power saving and entry/exit latency. The deeper the C-state, the more power saving and the longer latency a system can get.
- **Processor Performance state (P-state)**: again, in *C0* state, there are several sub-CPU performance states (P-States). In P-states, the CPU is working, but CPU voltage and frequency vary. The P-state is a very important power-saving feature.
- **Processor Throttling state (T-state)**: T-state is also a sub state of *C0*. It saves power by only changing CPU frequency. T-state is usually used to handle thermal event.
- **Sleeping state**: In *G1* state, it is divided into several sub state: $S1 \sim S4$. They differs in power saving, context preserving and sleep/wakeup latency. *S1* is lightweight sleep, with only CPU caches lost. *S2* is not supported currently. S3 has all context lost except system memory. S4 save context to disk and then lost all context. Deeper S-state is, more power saving and the longer latency system can get.

- **Device states (D-state)**: ACPI also defines power state for devices, i.e. *D0* $\sim$ *D3*. *D0* is working state and *D3* is power-off state. *D1* and *D2* are between *D0* and *D3*. *D0* $\sim$ *D3* differs in power saving, device context preserving and entry/exit latency.

Figure 1 in Len's paper [2] clearly illustrates the state relationship.

1.2 Virtualization model

Virtualization software is emerging in the open source world. Different virtualization model may have different implementation on power management, so it is better to check the virtualization model as below.

- **Hypervisor model**: virtual machine monitor (VMM) is a new layer below operation system and owns all the hardware. VMM not only needs to provide the normal virtualization functionality, e.g. CPU virtualization, memory virtualization, but also needs to provide the I/O device driver for every device.
- **Host-based model**: VMM is built upon a host operating system. All the platform hardware including CPU, memory, and I/O device, is owned by the host OS. In this model, VMM usually exists as a kernel module and can leverage much of the host OS functionality, e.g. I/O device driver, scheduler. Current KVM is host-based model.
- **Hybrid model**: VMM is a thin layer compared to the hypervisor model, which only covers basic virtualization functionality (CPU, memory, etc.), and leave I/O device to a privileged VM. This privileged VM provides I/O service to other VM through inter-VM communication. Xen [3] is hybrid model.

Meanwhile, with some I/O virtualization technology introduced, e.g. Intel® Virtualization Technology for Directed I/O, aka VT-d, the I/O device can be directly owned by a virtual machine.

1.3 Power Management in Virtualization

Power Management in virtualization basically has the following two types:

- **Virtual power management**: this means the power management within the virtual machine. This power management only has effects to the power state of VM and does not affect other VM or hypervisor/host OS. For example, virtual S3 sleep only brings VM into virtual sleep state, while hypervisor/host OS and other VM is still working. Virtual power management usually does not save real power, but sometimes it can affect real power management.
- **Real power management**: this means the power management in the global virtual environment, including the VMM/Host OS, and all VMs. This will operate on real hardware and save real power. The main guideline for global power management is that only the owner can do real power management operation to that device. And VMM/Host OS is responsible for coordinating the power management sequence.

This paper will elaborate how power management state (*Sx/Cx/Px/Dx*) is implemented for both virtual and real types.

2 Sleep States (*Sx*) Support

Linux currently supports *S1* (stand-by), *S3* (suspend-to-ram) and *S4* (suspend-to-disk). This section mainly discusses the *S3* and *S4* state support in the virtualization environment. *S1* and *S3* are similar, so the *S3* discussion can also apply to *S1*.

2.1 Virtual *S3*

Virtual *S3* is *S3* suspend/resume within a virtual machine, which is similar to native. When guest OSes see that the virtual platform has *S3* capability, it can start *S3* process either requested by user or forced by control tool under certain predefined condition (e.g. VM being idle for more than one hour). Firstly, the Guest OS freezes all processes and also write a wakeup vector to virtual ACPI FACS table. Then, the Guest OS saves all contexts, including I/O device context and CPU context. Finally, the Guest OS will issue hardware *S3* command, which is normally I/O port writing. VMM will capture the I/O port writing and handle the *S3* command by resetting the virtual CPU. The VM is now in virtual sleep state.

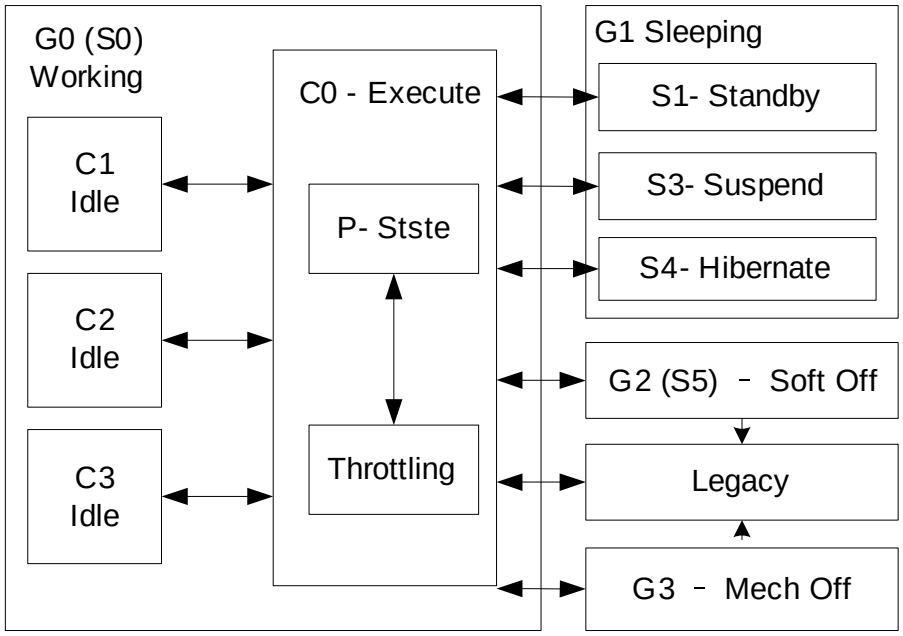

Figure 1: ACPI State Relationship

Guest OS *S3* wakeup is a reverse process. Firstly, VMM will put the virtual CPU into real mode, and start virtual CPU from guest BIOS POST code. BIOS POST will detect that it is a *S3* wakeup and thus jump to the *S3* wakeup vector stored in guest ACPI FACS table. The wakeup routine in turn will restore all CPU and I/O context and unfreeze all processes. Now the Guest OS resumes to working state.

From the above virtual *S3* suspend/resume process, it is easy to see that VMM needs the following work to support virtual *S3*:

- **Guest ACPI Table**: the ACPI DSDT table should have *_S3* package to tell guest OS that the virtual platform has *S3* capability, otherwise, guest OS won't even start *S3* sleep. Guest ACPI table can also have optional OEM-specific fields if required.

- **Guest BIOS POST Code**: Logic must be added here to detect the *S3* resume and get wakeup vector address from ACPI FACS table, and then jump to wakeup vector.

- ***S3* Command Interception**: Firstly, device model should emulate the ACPI PM1A control register, so that it can capture the S3 request. In KVM and Xen case, this can be done in QEMU side, and is normally implemented as a system I/O port. Secondly, to handle S3 request, VMM need to reset all virtual CPUs.

2.2 Real *S3* State

Unlike virtual *S3*, Real *S3* will put the whole system into sleep state, including VMM/Host OS and all the virtual machines. So it is more meaningful in terms of power saving.

Linux already has fundamental *S3* support, like to freeze/unfreeze processes, suspend/resume I/O devices, hotplug/unplug CPUs for SMP case, etc. to conduct a complete S3 suspend/resume process.

Real *S3* in virtualization also need similar sequence as above. The key difference is that system resources may be owned by different component. So the guideline is to ensure right owner to suspend/resume its owned resource.

Take Xen as an example. The suspend/resume operation must be coordinated among hypervisor, privileged VM and driver domain. Most I/O devices are owned by a privileged VM (domain0) and driver domain, so suspend/resume on those devices is mostly done in domain0 and driver domain. Then hypervisor will cover the rest:

- **Hypervisor owned devices**: APIC, PIC, UART, platform timers like PIT, etc. Hypervisor needs to suspend/resume those devices
- **CPU**: owned by hypervisor, and thus managed here
- **Wakeup routine**: At wakeup, hypervisor need to be the first one to get control, so wakeup routine is also provided by hypervisor.
- **ACPI PM1x control register**: Major ACPI sleep logic is covered by domain0 with the only exception of PM1x control register. Domain0 will notify hypervisor at the place where it normally writes to PM1x register. Then hypervisor covers the above work and write to this register at the final stage, which means a physical *S3* sleep.

For the driver domain that is assigned with physically I/O device, hypervisor will notify these domains to do virtual *S3* first, so that these domains will power off their I/O device before domain0 starts its sleep sequence.

Figure 2 illustrates the Xen Real *S3* sequence.

2.3 Virtual *S4* State and Real *S4* State

Virtual *S4* is suspend-to-disk within virtual machine. Guest OS is responsible to save all contexts (CPU, I/O device, memory) to disk and enter sleep state. Virtual *S4* is a useful feature because it can reduce guest OS booting time.

From the VMM point of view, virtual *S4* support implementation is similar as virtual *S3*. The guest ACPI also needs to export *S4* capability and VMM needs to capture the *S4* request. The major difference is how VMM handles the *S3/S4* request. In *S3*, VMM needs resetting VCPU in *S3* and jumps to wakeup vector when VM resuming. In *S4*, VMM only needs to destroy the VM since VMM doesn't need to preserve the VM memory. To resume from *S4*, user can recreate the VM with the previous disk image, the guest OS will know that it is *S4* resume and start resuming from *S4*.

Real *S4* state support is also similar as native *S4* state. For host-based model, it can leverage host OS *S4* support directly. But it's more complex in a hybrid model like Xen. The key design issue is how to coordinate hypervisor and domain0 along the suspend process. For example, disk driver can be only suspended after VMM dumps its own memory into disk. Then once hypervisor finishes its memory dump, later change on virtual CPU context of domain0 can not be saved any more. After wakeup, both domain0 and hypervisor memory image need to be restored and sequence is important here. This is still an open question.

3 Processor Power States (*Cx*) support

Processor power states, while in the *G0* working state, generally refer to active or idle state on the CPU. *C0* stands for a normal power state where CPU dispatches and executes instructions, and *C1*, *C2* $\cdots$ *Cn* indicates low-power idle states where no instructions are executed and power consumption is reduced to a different level. Generally speaking, a larger value of *Cx* brings greater power savings, at the same time adds longer entry/exit latency. It's important for OSPM to understand ability and implication of each C-state, and then define appropriate policy to suit activities of the time:

- Methods to trigger specific C-state
- Worst case latency to enter/exit C-state
- Average power consumption at given C-state

Progressive policy may hurt some components which don't tolerate big delay, while conservative policy makes less use of power-saving capability provided by hardware. For example, OSPM should be aware that cache coherency is not maintained by the processor when in *C3* state, and thus needs to manually flush cache before entering when in SMP environment. Based on different hardware implementation, TSC may be stopped and so does LAPIC timer interrupt. When *Cx* comes into virtualization, things become more interesting.

3.1 Virtual C-states

Virtual C-states are presented to VM as a 'virtual' power capability on 'virtual' processor. The straight-forward effect of virtual C-states is to exit virtual processor from scheduler when *Cx* is entered, and to wake virtual processor back to scheduler upon break event. Since virtual processor is 'virtual' context created and managed by VMM, transition among virtual C-states have nothing

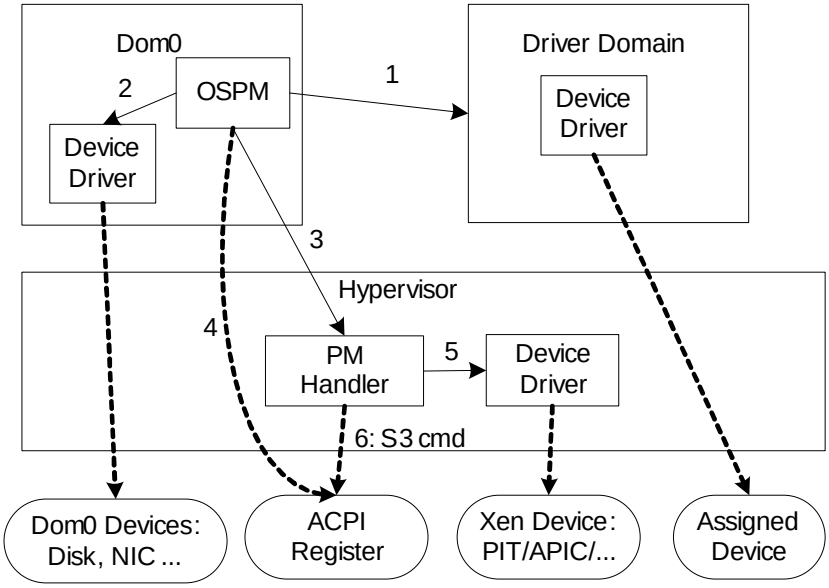

Figure 2: Xen Real *S3* sequence

to do with power state on real processors, but does have the ability to provide useful hints in some cases.

The way to implement virtual C-states can vary upon the virtualization model. For example, a hardware-assisted guest may be presented with C-states capability fully conforming to ACPI specification, while a para-virtualized guest can simply take quick hyper-call to request. Basically it doesn't make much sense to differentiate among *C1*, *C2*$\cdots$*Cn* in a virtualization world, but it may be useful to some cases. One direct case is to test processor power management logic of a given OSPM, or even to try whether some newer C-state is a meaningful model before hardware is ready. Another interesting usage would be to help physical C-state governor for right decision, since virtual C-state request reveals activities within a guest.

3.1.1 Para-virtualized guest

para-virtualized guest is a modified guest which can cooprate with VMM to improve performance. virtual C-state for a para-virtualized guest just borrows the term from ACPI, but no need to bind with any ACPI context. A simple policy can just provide 'active' and 'sleep' categories for a virtual processor, without differentiation about *C1*$\cdots$*Cn*. When idle thread is scheduled without anything to handle, time events in the near future are walked for nearest interval which is then taken as parameter of sleep hyper-call issued to VMM. Then VMM drops the virtual CPU from the run-queue and may wake it up later upon any break event (like interrupt) or specified interval timeout. A perfect match on a tick-less time model! Since it's more like the normal 'HALT' instruction usage, the policy is simple which is tightly coupled with time sub-system.

It's also easy to extend para-virtualized guest with more fine-grained processor power states, by extending above hyper-calls. Such hyper-call based interface can be hooked into generic Linux processor power management infrastructure, with common policies unchanged but a different low-level power control interface added.

3.1.2 Hardware-assisted guest

Hardware-assisted guest is the unmodified guest with hardware (e.g. Intel VT or AMD-V) support. Not like a para-virtualized guest who poses changes within the guest, virtual platform (i.e., device model) needs export exact control interface conforming to ACPI spec and emulate desired effect as what hardware-assisted guest expects. By providing the same processor power management capability, no change is required within hardware-assisted guest.

- **Virtual *C2*** – an ability provided by chipset which

controls clock input signal. First device model needs to construct correct ACPI table to expose related information, including trigger approach, latency and power consumption as what ACPI spec defines. ACPI FADT table contains fixed format information, like P_LVL2 command register for trigger. Recent ACPI version also adds a more generic object _CST to describe all C-state information, e.g. C state dependency and mwait extension. Device model may want to provide both methods if taken as a test environment.

Device model then needs to send a notification to VMM after detecting virtual C2 request from guest. As acceleration, Cx information can be registered to VMM and then VMM can handle directly. Actually, for virtual C2 state, device model doesn't need to be involved at run time. C2 is defined as a low-power idle state with bus snooped and cache coherency maintained. Basic virtual MMU management and DMA emulation have already ensured this effect at given time.

- **Virtual *C3*** – almost the same as virtual *C2*, P_LVL3 or _CST describe the basic information. But virtual *C3* also affects device model besides virtual processor. Device model needs to provide PM2_CNT.ARB_DIS which disables bus master cycles and thus DMA activities. PM1x_STS.BM_STS, an optional feature of chipset virtualization, reveals bus activity status which is a good hint for OSPM to choose *C2* or *C3*. More importantly, PM1x_CNT.BM_RLD provides option to take bus master requests as break event to exit *C3*. To provide correct emulation, tight cooperation between device model and VMM is required which brings overhead. So it's reasonable for device model to give up such support, if not aimed to test OSPM behavior under *C3*.

- **Deeper virtual *Cx*** – similar as *C3*, and more chipset logic virtualization are required.

3.2 Real C-states

VMM takes ownership of physical CPUs and thus is required to provide physical C-states management for 'real' power saving. The way to retrieve C-states information and conduct transition is similar to what today OSPM does according to ACPI spec. For a host based VMM like KVM, those control logic has been there in the host environment and nothing needs to be changed. Then, for a hybrid VMM like Xen, domain0 can parse and register C-state information to hypervisor which is equipped with necessary low-level control infrastructure.

There are some interesting implementation approaches . For example, VMM can take a virtual *Cx* request into consideration. Normally guest activities occupy major portion of cpu cycles which can then be taken as a useful factor for C-state decision. VMM may then track the virtual C-state requests from different guests, which represent the real activities on given CPU. That info can be hooked into existing governors to help make better decisions. For example:

- Never issue a C-x transition if no guest has such virtual C-x request pending
- Only issue a C-x transition only if all guests have same virtual C-x requests
- Pick the C-x with most virtual C-x requests in the given period

Of course, the above is very rough and may not result in a really efficient power saving model. For example, one guest with poor C-state support may prevent the whole system from entering a deeper state even when condition satisfies. But it does be a good area for research to leverage guest policies since different OS may have different policy for its specific workload.

4 Processor Performance States (*Px*) Support

P-states provide OSPM an opportunity to change both frequency and voltage on a given processor at run-time, which thus brings more efficient power-saving ability. Current Linux has several sets of governors, which can be user-cooperative, static, or on-demand style. P-states within the virtualization world are basically similar to the above C-states discussion in many concepts, and thus only its specialties are described below.

4.1 Virtual P-states

Frequency change on a real processor has the net effect to slow the execution flow, while voltage change is at

fundamental level to lower power consumption. When coming to virtual processor, voltage is a no-op but frequency does have useful implication to the scheduler. Penalty from scheduler has similar slow effect as frequency change. Actually we can easily plug virtual P-state requests into schedule information, for example:

- Half its weight in a weight-based scheduler
- Lower its priority in a priority-based scheduler

Furthermore, scheduler may bind penalty level to different virtual P-state, and export this information to guest via virtual platform. Virtual platform may take this info and construct exact P-states to be presented to guest. For example, *P1* and *P2* can be presented if scheduler has two penalty levels defined. These setup the bridge between virtual P-states and scheduler hints. Based on this infrastructure, VMM is aware of guest requirement and then grant cycles more efficiently to guest with more urgent workload.

4.2 Real P-states

Similar as real C-states for virtualization, we can either reuse native policy or add virtualization hints. One interesting extension is based on user space governor. We can connect together all guest user space governors and have one governor act as the server to collect that information. This server can be a user space governor in host for a host-based VMM, or in privileged VM for hybrid VMM. Then, this user space governor can incorporate decisions from other user space governors and then make a final one. Another good point for this approach is that hybrid VMM can directly follow request from privileged VM by a simple "follow" policy.

5 Device Power States (*Dx*) Support

Devices consume another major portion of power supply, and thus power feature on devices also plays an important role. Some buses, like PCI, have well-defined power management feature for devices, and ACPI covers the rest if missing. Power state transition for a given device can be triggered in either a passive or active way. When OSPM conducts a system level power state transition, like *S3*/*S4*, all devices are forced to enter appropriate low power state. OSPM can also introduce active on-demand device power management at run-time, on some device if inactive for some period. Carefulness must be taken to ensure power state change of one node does not affect others with dependency. For example, all the nodes on a waken path have to satisfy minimal power requirement of that wake method.

5.1 Virtual D-states

Devices seen by a guest are basically split into three categories: emulated, para-virtualized, and direct-assigned. Direct-assigned devices are real with nothing different regarding D-states. Emulated and para-virtualized are physically absent, and thus device power states on them are also virtual.

Normally, real device class defines what subset of capabilities are available at each power level. Then, by choosing the appropriate power state matching functional requirement at the time, OSPM can request device switching to that state for direct power saving at the electrical level. Virtual devices, instead, are completely software logics either emulated as a real device or para-virtualized as a new device type. So virtual D-states normally show as reduction of workload, which has indirect effect on processor power consumption and thus also contributes to power saving.

For emulated devices, the device model presents exact same logic and thus D-states definition as a real one. Para-virtualized devices normally consist of front-end and back-end drivers, and connection states between the pair can represent the virtual D-states. Both device model and back-end need to dispatch requests from guest, and then handle with desired result back. Timer, callback, and kernel thread, etc. are possible components to make such process efficient. As a result of virtual D-states change, such resources may be frozen or even freed to reduce workload imposed on the physical processor. For example, the front-end driver may change connection state to 'disconnected' when OSPM in guest requests a D3 state transition. Then, back-end driver can stop the dispatch thread to avoid any unnecessary activity caused in the idle phase. Same policy also applies to device model which may, for example, stop timer for periodically screen update.

Virtual bus power state can be treated with same policy as virtual device power state, and in most time may be just a no-op if virtual bus only consists of function calls.

5.2 Real D-states

Real device power states management in virtualization case are a bit complex, especially when device may be direct assigned to guests (known as a driver domain). To make this area clear, we first show the case without driver domain, and then unveil tricky issues when the later is concerned.

5.2.1 Basic virtualization environment

Basic virtualization model have all physical devices owned by one privileged component, say host Linux for KVM and domain-0 for Xen. OSPM of that privileged guy deploys policies and takes control of device power state transitions. Device model or back-end driver are clients on top of related physical devices, and their requests are counted into OSPM's statistics for given device automatically. So there's nothing different to existing OSPM.

For example, OSPM may not place disk into deeper D-states when device model or back-end driver is still busy handling disk requests from guest which adds to the workload on real disk.

As comparison to the OSPM within guests, we refer to this special OSPM as the "dominate OSPM." Also dominator is alias to above host Linux and domain-0 in below context for clear.

5.2.2 Driver domains

Driver domains are guests with some real devices assigned exclusively, to either balance the I/O virtualization bottleneck or simply speed the guest directly. The fact that OSPM needs to care about the device dependency causes a mismatch on this model: dominate OSPM with local knowledge needs to cover device dependencies across multiple running environments.

A simple case (Figure 3) is to assign *P2* under PCI Bridge1 to guest GA, with the rest still owned by dominator. Say an on-demand D-states governor is active in the dominate OSPM, and all devices under *Bridge1* except *P2* have been placed into *D3*. Since all the devices on bus 1 are inactive now based on local knowledge, dominate OSPM may further decide to lower power

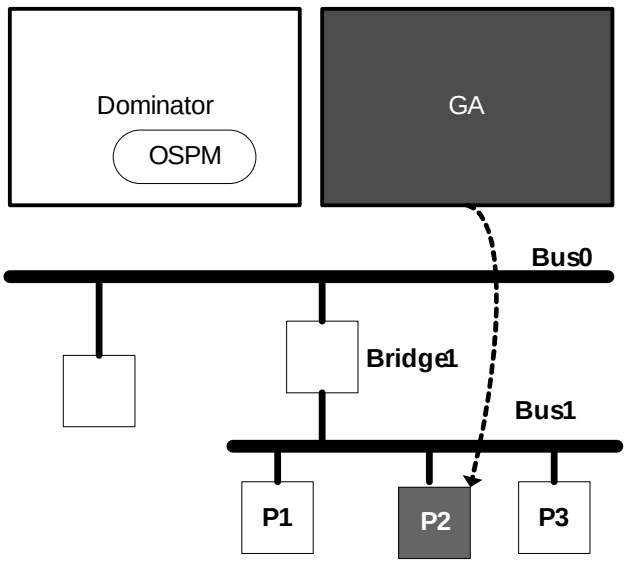

Figure 3: A simple case

voltage and stop clock on bus 1 by conducting *Bridge1* into a deeper power state. Devil take it! *P2* can never work now, and GA has to deal with a dead device without response.

Then, the idea is simple to kick this issue: extend local dominate OSPM to construct full device tree across all domains. The implication is that on-demand device power governor can't simply depend on in-kernel statistics, and hook should be allowed from other components. Figure 4 is one example of such extension:

Device assignment means grant of port I/O, MMIO, and interrupt in substance, but the way to find assigned device is actually virtualized. For example, PCI device discovery is done by PCI configuration space access, which is virtualized in all cases as part of virtual platform. That's the trick of how the above infrastructure works. For hardware-assisted guest, device model intercepts access by traditional 0xcf8/0xcfc or memory mapped style. Para-virtualized guest can have a PCI frontend/backend pair to abstract PCI configuration space operation, like already provided by today's Xen. Based on this reality, device model or PCI backend can be good place to reveal device activity if owned by other guests, since standard power state transition is done by PCI configuration space access as defined by PCI spec. Then based on hint from both in-kernel and other virtualization related components, dominate OSPM can now precisely decide when to idle a parent node if with child nodes shared among guests.

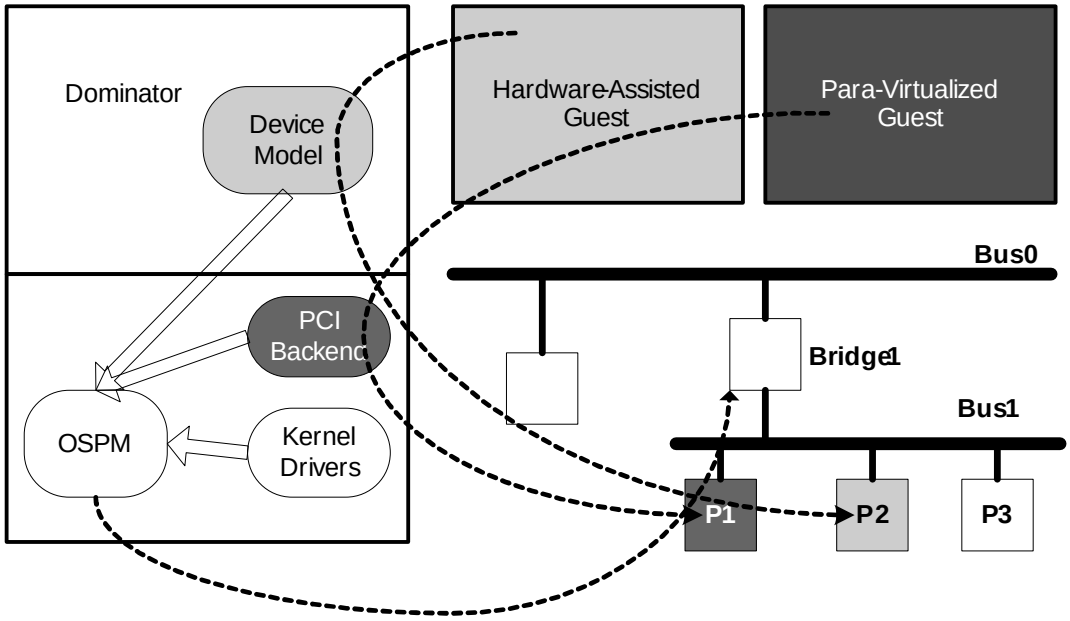

Figure 4: The extended case

However, when another bus type is concerned without explicit power definition, it's more complex to handle device dependency. For example, for devices with power information provided by ACPI, the control method is completely encapsulated within ACPI AML code. Then the way to intercept power state change has to be a case specific approach, based on ACPI internal knowledge. Fortunately, most of the time only PCI devices are preferred regarding the device assignment.

6 Current Status

Right now our work on this area is mainly carried on Xen. Virtual *S3*/*S4* to hardware-assisted guest has been supported with some extension to ACPI component within QEMU. This should also apply to other VMM software with same hardware-assisted support.

Real *S3* support is also ready. Real S3 stability relies on the quality of Linux S3 support, since domain0 as a Linux takes most responsibility with the only exception at final trigger point. Some linux S3 issues are met. For example, SATA driver with AHCI mode has stability issue on 2.6.18 which unfortunately is the domain0 kernel version at the time. Another example is the VGA resume. Ideally, real systems that support Linux should restore video in the BIOS. Real native Linux graphics drivers should also restore video when they are used. If it does not work, you can find some workaround in `documentation/power/video.txt`. The positive side is that Linux S3 support is more and more stable as time goes by. Real S4 support has not been started yet.

Both virtual and real Cx/Px/Tx/Dx supports are in development, which are areas with many possibilities worthy of investigation. Efficient power management policies covering both virtual and real activities are very important to power saving in a run-time virtualization environment. Forenamed sections are some early findings along with this investigation, and surely we can anticipate more fun from this area in the future.

References

[1] "Advanced Configuration & Power Specification," Revision 3.0b, 2006, Hewlett-Packard, Intel, Microsoft, Phoenix, Toshiba.
`http://www.acpi.info`

[2] "ACPI in Linux," L. Brown, A. Keshavamurthy, S. Li, R. Moore, V. Pallipadi, L. Yu, In *Proceedings of the Linux Symposium* (OLS), 2005.

[3] "Xen 3.0 and the Art of Virtualization," I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D.

Magenheimer, J. Nakajima, and A. Mallick, in *Proceedings of the Linux Symposium* (OLS), 2005.

Proceedings of the Linux Symposium

Volume One

June 27th–30th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Dirk Hohndel, *Intel*
Martin Bligh, *Google*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*