Virtualizing Performance Counters

Benjamin Serebrin and Daniel Hecht

VMware
{serebrin,dhecht}@vmware.com

Abstract. Virtual machines are becoming commonplace as a stable and flexible platform to run many workloads. As developers continue to move more workloads into virtual environments, they need ways to analyze the performance characteristics of those workloads. However, performance efforts can be hindered because the standard profiling tools like VTune and the Linux Performance Counter Subsystem do not work in most modern hypervisors. These tools rely on CPUs' hardware performance counters, which are not currently exposed to the guests by most hypervisors. This work discusses the challenges of performance counters due to the trap and emulate method of virtualization and the time sharing of physical CPUs among multiple virtual CPUs. We propose an approach to address these issues to provide useful and intuitive information about guest performance and the relative costs of virtualization overheads.

1 Introduction

Virtualization is a method to decouple physical hardware from an operating system by running the guest OS in a virtual machine, or VM. Virtualization has found uses in both development, to isolate systems-under-test, and in data centers, to provide server consolidation, migration, and manageability functions. Workloads are increasingly likely to be virtualized. Many companies are using remote desktop viewing products to replace developers' and users' dedicated desktop machines. With these trends, developers face an increasing need to use native performance tools like VTune[13], CodeAnalyst[3], and the Linux Performance Counter Subsystem[6] inside VCPUs.

VMware is investigating the virtualization of hardware performance counters for its future products. This paper discusses the typical use cases for the counters and the ways to properly represent a useful view of performance events. The typical uses of virtualization present several challenges not present on non-virtualized (native) systems. For example, a virtual CPU (VCPU) does not occupy a physical CPU (PCPU) 100% of the time, which breaks many assumptions a profiler may make about the passage of time. Sharing the PCPU with multiple VMs requires that the hypervisor not only context switch the hardware counter resources, but also adjust some of the results to match what the profiler is expecting.

These event count adjustments are also required to expose the resource and time consumption of emulated instructions, and to match the semantics of a

M. Alexander et al. (Eds.): Euro-Par 2011 Workshops, Part I, LNCS 7155, pp. 223–233, 2012.

small set of counters that relate to instruction retirement. For example, a hypervisor often completes a trapped privileged instruction on behalf of the guest. Without additional hypervisor bookkeeping, this emulation would not appear as having consumed execution resources or retired any instructions if the counters are paused while the hypervisor is running. Conversely, if counters are allowed to freely run in both contexts, confusing results may arise when hypervisor events are counted. Our work seeks to find the best mix of performance counter emulations to represent performance events in a way that supports existing profiling tools.

The hypervisor strives to provide a virtual CPU as similar as possible to physical hardware; it should present virtualized performance counters that enable a guest to profile itself as well as it could on a native system. This work treats the hypervisor itself as an opaque extension of the CPU with respect to the in-guest profiling system; this enables the hypervisor to agnostically support all profilers that use the standard CPU interface to performance counters.

Enabling performance counter use in guests allows profilers to use hardware performance counters to measure metrics including cache misses, translation lookaside buffer (TLB) pressure, and instructions per cycle (IPC). These profiling tools produce per-process and system metrics based on sampled hardware counts to indicate where performance improvement opportunities exist. High performance computing developers frequently use such performance tools to find bottlenecks and inefficiencies. Virtualization is under discussion in HPC contexts to provide abstraction, migration, and reliability, and hypervisor vendors must support the same performance monitoring capabilities that developers use to tune native systems.

2 x86 Performance Counter Hardware

Intel[8] and AMD[1] provide similar interfaces to their performance counting hardware. Each CPU has its own set of performance counters and performance event select registers. The event select register is used to specify which microarchitectural event is to be counted, and contains bits to enable, filter the count results, and raise interrupts if the counter overflows from negative to positive. Typical modern hardware has between 2 and 8 general purpose counters and up to 3 fixed counters, each dedicated to a single event.

The encodings of events are generation-specific, though Intel has defined a small set of events that are encoded consistently across generations, and similar events on AMD appear consistent as well. Events common across many architectures include cycle counts (relative to core cycles and to constant-rate cycles), TLB accesses and misses, last-level cache accesses and misses, and instruction and branch retired counts. In addition to these common events, each CPU generation has its own assortment of architecture-specific events including store-to-load forwarding failure counts and functional unit stall events. AMD and Intel each have mechanisms to enable and disable individual counters during the state transition between the hypervisor's own code and running the guest.

A typical usage of the performance counters could include configuring Event Select 0 to count Last-Level Cache misses in all privilege levels with the overflow interrupt disabled and configuring Event Select 1 to count Last-Level Cache accesses with identical privilege and interrupt settings. A profiler then samples the Event Counts 0 and 1 and calculates per-sample period differences to track the ratio of cache misses to accesses. In addition, the interrupt facility of the hardware counters can be enabled to cause interrupts after a set number of events. In this example, setting the Event Select's interrupt-enable bit and setting the corresponding counter to -10,000 would cause the hardware to raise an interrupt after the 10,000th cache miss.

Extensions such as PEBS (Precise Event-Based Sampling) and LWP (Lightweight Profiling) are not discussed in detail in this paper, but may be virtualized with similar methods. However, each of these has memory-access characteristics that present more virtualization challenges than the simple Model-Specific Register (MSR) interface of the core event counters. Uncore or Northbridge counters are shared among multiple PCPUs and thus are less amenable to time-multiplexing.

3 Profilers

Profilers commonly use a sampling mode where one or more events are allowed to raise interrupts after a specified number of counts. The user or profiler sets the overflow limit to achieve a desired approximate frequency of events, based on an estimated expected rate of events. For example, if 1000 TLB misses occur per second on average, and a 10Hz sampling rate is desired, the event counter is initialized to -100.

The profilers are given knowledge of a binary's code layout and symbols and are often integrated with an IDE programming environment like Visual Studio or Eclipse. They run one or multiple passes of a program and record the instruction pointer and possibly the call stack information in place at the time of a performance counter interrupt. The profilers present a visualization of the average cost of a given function or even an individual instruction based on this information.

A programmer commonly writes a program and runs it under the profiler, providing a typical input set. The programmer can iteratively apply optimizations based on profiler results.

4 Time-Sharing Physical CPUs

Many datacenters perform CPU overcommittment using hypervisors, running multiple VMs on a single computer where the total VCPU count exceeds the total number of PCPUs. The hypervisor must share PCPUs among all the VCPUs, giving each VCPU a fraction of the total runtime of the system. The sharing of hardware resources requires the hypervisor to apply heuristics to enable guest operating systems to accurately keep track of absolute time, often called wall-clock time.[14]

The guest operating system wall-clock should track absolute time over the long term. To achieve this, while the VM is descheduled, VMware's virtual timer devices that are used by the guest operating system for timekeeping are allowed to fall behind real time and later catch up faster than real time when the VM is rescheduled. This way, over the longer term, these devices track absolute real time. Profilers, on the other hand, are more concerned with relative time differences over the short term, and want to count only the time that the VCPU is scheduled on a PCPU.

This tension over the desired semantics of a timer device requires the hypervisor to carefully trade off keeping a guest's notion of wall-clock time correct and giving a notion of time appropriate for profilers' use. Both Intel and AMD CPUs provide an event called core cycles not halted, which tracks the CPU cycle count independently of wall-clock time. CPU frequency can increase or decrease due to power saving modes, and CPU cycles can stop entirely if the OS has executed the `HLT` instruction. The notion of core cycles not halted is thus a convenient hardware interface that can be extended for profiling in a virtual environment. The hypervisor can define core cycles not halted to count only core clock cycles when the VCPU is in context on a PCPU, including time spent in the hypervisor on that VCPU's behalf. Fortunately, the use of core cycles not halted appears to be common practice in profilers, and our extension of its meaning is consistent with common calculations of events per unhalted cycle ratios.

The `RDTSC` instruction is a common source of total elapsed time. However, an instructions per cycle calculation that uses `RDTSC` results for elapsed time as its denominator might under-count the IPC. In this case, the hypervisor is actively managing the `RDTSC` instruction to track absolute real time. The more correct method would be to measure instructions retired in one performance counter while measuring core-cycles not halted in another. Fortunately, there is incentive on native systems for that practice, as most modern processors support some form of hardware-mediated cycle speed boosting that the OS does not control. Boosting provides a similar distortion as hypervisor time manipulation when `RDTSC` is used as a cycle count.

Sharing hardware leads to other, less direct effects. Just as multiple processes may compete for cache and other resources, multiple VCPUs and other unrelated hypervisor threads that share a physical core can pollute each other's caches, branch predictors, TLBs, and other microarchitectural state. This work intentionally avoids attempting to condition these types of counters, both due to the difficulty of properly recalculating the non-sharing values of such dynamic counts, and due to a desire to appropriately show the effects of sharing resources. While a program may not be causing cache misses itself, it may still experience them in a time-shared machine, and the programmer could benefit from such knowledge.

The hypervisor context switches all relevant CPU state when each VCPU is scheduled and descheduled. To virtualize performance counters as we describe, the hypervisor must context switch the active performance counter state, in

addition to the context switching of general purpose registers and control state. This serves to time-multiplex the CPU and performance counter hardware resources and guarantee that virtual counters do not advance while that VCPU is out of context. The context switching of the counter state satisfies our extended definition of unhalted core-cycles.

5 Trapping and Emulation

When a non-privileged guest instruction is executed by the physical CPU, the guest is said to be in *direct execution* mode. Modern hypervisors trap and emulate[12] to handle privileged guest instructions and events. In well-tuned systems, the rate of traps is low and guests spend most of their time in direct execution. However, a guest may execute a trapped instruction, such as the `CPUID` instruction, which is then intercepted by the hypervisor. The hypervisor decodes and emulates the instruction, and resumes with direct execution beginning at the next guest instruction. Other mandatory traps include `IN` and `OUT` instructions, page faults that are induced by lazy context evaluation in shadow or nested paging modes, and accesses to virtual devices.

If a hypervisor were to naïvely pause the virtual counters when exiting direct execution, an emulated instruction would never cause any counters to increment. Conversely, if the hypervisor were to allow all counters to continue running while emulating the instruction, the instructions retired count would match the number of hypervisor instructions executed. For example, if the following code snippet were in an inner loop, a non-virtualized profiler would discover that many cycles were spent executing the `CPUID` instruction (which is fairly expensive at approximately 100 cycles on modern CPUs) and that IPC is very low. However, a hypervisor that paused virtual counters would fail to increment cycles or even instruction retired counts for the `CPUID` instruction, and the profiler would interpret the event counts as saying the `CPUID` instruction had zero cost. Alternatively, if all counters increment freely during emulation of the `CPUID` instruction, IPC may appear to be high since most of the hypervisor instructions used to emulate `CPUID` will be normal high-throughput instructions. This design choice would also hide the cost of executing `CPUID` in a VM. Our work aims to categorize ways to appropriately and efficiently emulate performance events in such cases.

```
int popcnt(int i) {
   if (cpuid(1).ecx & POPCNT_MASK != 0) {
      return __builtin_popcnt(i);
   } else {
      return SoftwarePopcnt(i);
   }
}
```

6 Speculative and Non-speculative Events

Many performance events are subject to run-to-run variation due to processor speculation, varying cache and branch predictor temperature, variable cache miss costs, and other non-deterministic effects. These *speculative* [1] events include cache, branch predictor, and TLB statistics as well as microarchitecture-specific events. A cache miss counter, for instance, could experience run-to-run variation due to execution of mispredicted code, OS context switch costs, cache misses due to thread migration to different cores, and memory bandwidth competition due to other cores.

Other events can be considered deterministic and *non-speculative*: for a given program execution, the counts of retired instructions and branches can be expected to be repeatable and determined from an in-order execution of the program. For example, after a 3-instruction code loop that executes 1000 times, the retired branch instruction event should report 1000 branches and the retired instruction event should report 3000 instructions.

7 Combining Native and Emulated Performance Counters

As discussed above, one approach is to allow the counters to run during execution of the inner portions of the emulation code, and to pause the counters only on context switches away from the current VCPU. Another approach pauses all counters at the boundary between hypervisor and hardware execution of guest code.[4] Finally, a third approach emulates the hardware counters to attempt to represent the microarchitecture's counts for a small subset of events.[5]

This paper proposes a hybrid approach: for non-speculative events, the emulation code will ensure correctness, while speculative counters will present a view of the hypervisor's effect on hardware even for emulated guest instructions and events.

When the hypervisor is emulating one or more guest instructions, it has full knowledge of the counts of non-speculative guest events and can increment the counters. However, it is impractical to provide a CPU simulator in order to properly represent cache miss rates, TLB hits, cycle counts, and other speculative events. Instead, this work allows speculative events to count both in the guest context and during hypervisor emulation of the instruction. As discussed above, if the hypervisor switches context to another VCPU, the virtual counters are paused. When the current VCPU context is restored, the virtual counters resume.

Our approach gives the guest a glimpse into the costs of the hypervisor's implementation, but does not propose to expose hypervisor addresses or symbols. Instead, hypervisor effort for emulation is attributed to the instruction that required the emulation by allowing the non-speculative events to continue counting in the hypervisor. In the code example above, our approach causes `CPUID` to appear to consume the number of cycles that were required to emulate the instruction, and to increment the retired instruction count by one.

This design leads to some interesting surprises. For example, a natively-executed `CPUID` instruction never causes TLB misses or cache misses. However, the emulation code does require memory accesses and is likely to induce both TLB and cache events. We deliberately pass such counts through to the guest. This kind of induced event incrementing is visible only to a spot inspection of a particular instruction. A typical profiler must tolerate imprecision including the variable number of cycles a performance counter interrupt can require to arrive. Therefore, such surprising events like unexpected cache misses on non-memory instructions, which do not occur on native hardware, are unlikely to confuse the profiler's results.

Our design results in approximate correctness: expensive events do appear expensive when viewed in the profiler, whether the instruction causing that event was trapped-and-emulated or executed by hardware.

7.1 Hybrid Performance Counter Example

Figure 1 demonstrates how the various classes of performance events can behave under our design. Instructions retired is a non-speculative event, so does not increment while the VCPU is descheduled. However, at the point indicated by the arrow, the hypervisor has emulated one or more instructions and has incremented the counter accordingly. The hypervisor (VMM) runs between the VMexit and VMentry times. TLB accesses is a speculative event and is allowed to run during that time. Cycles not halted, while not related to speculative execution, is treated in the same manner. All VCPU counters are stopped when the VCPU is descheduled.

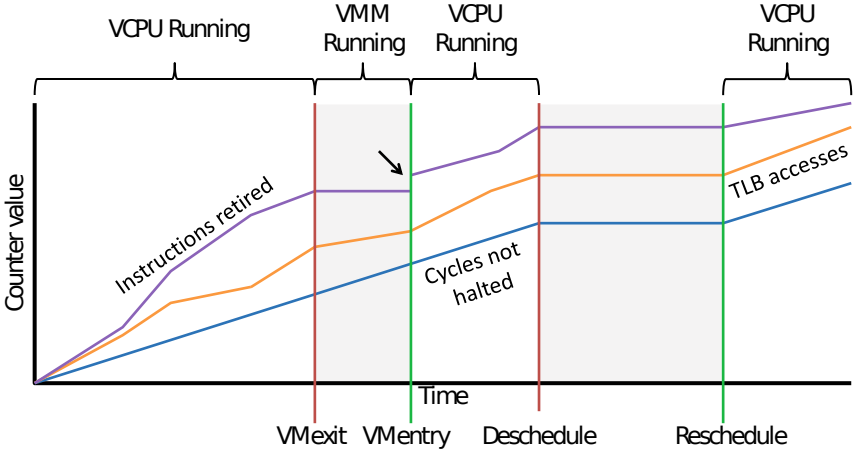

Fig. 1. Example graph of speculative and nonspeculative counters

8 Performance Counter Hardware Virtualization Methods

VMware hypervisors have historically largely avoided paravirtualized interfaces to basic hardware features in an effort to focus performance optimization at supporting a wide range of virtualization-unaware guests. In general, guest accesses to privileged state are trapped and emulated by the hypervisor. The hypervisor's handling of guest performance counter resource accesses follows this pattern: each guest performance counter MSR access is trapped and proxied, largely unaltered, to the underlying hardware.[1] A guest performance counter that is configured to raise an interrupt will cause the hypervisor to enable the interrupt on the corresponding hardware resource. The hypervisor will trap the resulting performance monitor interrupt and forward it synchronously to the guest.

The hypervisor configures the hardware performance counters with a few small differences to enable the hypervisor to share performance counters with other hypervisor software services, and to enable the guest-only filtering capabilities of the hardware when appropriate for the event type.

9 Discussion

An open item of discussion for this workshop is to explore whether these design choices are correct and adequate.

One question is how the hypervisor should properly filter events that a profiler might use in a ratio. If the profiler mixes speculative and non-speculative events in the same ratio, the speculative events will have been allowed to increment during hypervisor emulation code, while the non-speculative events will count only guest events. Example ratios include TLB misses or accesses per retired instruction. A hypervisor cannot predict whether the profiler or user wants to include only events incurred while running guest code, but a modification to our design could assume the use of retired instruction events implies such a ratio and thus conditionally pause speculative event counts during emulation.

Generally, it is expected that unmodified profiling tools will function correctly under this virtualization scheme. Directed tests running natively and in VMs observe performance counter results that match within close tolerances, and it is expected that variances due to the timesharing and virtualization will cause similarly minor disruptions to profiling tools' results. The user bears some responsibility for understanding profiler results in the native case, and some additional understanding about the nature of virtualized workloads will be vital to effective performance tuning.

Certain phases of a workload should be minimally impacted by virtualization, while others may experience a range of performance effects. For example, a computation-heavy workload phase that does not pressure the TLB significantly

[1] Optimizations to allow the guest to write some MSRs directly are not the focus of this paper. The hypervisor can usually allow the guest to write the count MSRs directly, but often must mediate access to configuration and interrupt control registers.

should provide the same profile results when run in a VM or natively. The profiler would be expected to show significant profile differences if, for example, the VM's memory is misconfigured and resides on an inappropriate memory controller. The shape of the profile for other workloads can be expected to change under virtualization. For example, using hardware page table virtualization (Intel EPT or AMD Nested Paging) can increase the cost of a TLB miss.[2] A garbage-collected workload with a large working set could experience greater performance degradations while virtualized, and the costs will be visible in the profile.

The profilers discussed here are often considered vertical profilers[7], in that they have sampling and symbol visibility to multiple layers of the system, including hardware, operating system, various middle layers, language virtual machine, and application. While software counters can be used for the upper layers of the system, the hardware's own performance counters are typically used for measuring the hardware layer. This work can be considered to support vertical profiling by extending the definition of hardware to include the CPU emulation efforts of the hypervisor. This definition serves to abstract away specific knowledge of the hypervisor from the guest profiler while exposing the related costs, in a manner analogous to the level of visibility into the CPU's hardware itself.

10 Future Work

This work focuses on enabling existing unmodified performance tools. By inferring the appropriate way to emulate speculative and non-speculative counters, we assume knowledge of the profiler's intentions. A virtualization-aware profiler could express its interest in how a hypervisor should emulate its counters.

Analogous to the commonality of many of the basic microarchitectural events, hypervisors could expose synthesized events that are common to most hypervisor implementations, especially events that pertain to implementation of CPU emulations. For example, a shadow-paging hypervisor could expose the number of hidden page faults that required hypervisor intervention for events like accessed and dirty bit setting. A hypervisor in any paging mode, including hardware-supported nested paging, can expose an event containing the number of hidden page faults due to lazy population of guest memory or copy-on-write collisions.

Timeslicing a machine is a fertile area of exploration for adding performance metrics. A guest may be interested in estimating the number of cycles stolen due to resource sharing (for example, to understand video frame rates or other time-sensitive measurements). Disk events, hypervisor lock statistics, and NUMA migration counts are other potential events that could provide insight into the system.

11 Related Work

Generally, existing profiling tools that can be used with virtual machine environments are modified and either run on the host or have some cooperation from and communication with the hypervisor. [9] discusses a system-wide profiling

framework, Xenoprof, implemented in Xen. It includes a hypervisor component and a domain level component that coordinate to provide a system-wide profiler. OProfile has been ported to the Xenoprof interface. [16] allows a Linux host running the KVM VMM to collect guest OS statistics using the Linux perf utility. [15] can count hardware performance events on an ESX hypervisor.

[4], [5] and [11] discuss recent performance counter extensions to KVM and Xen, respectively. [4] and [5] discuss *guest-wide profiling*, which is similar in intent to our approach. [4] and [5] also discuss two choices for multiplexing the performance counter hardware: *CPU switch* and *domain switch*. [11] describes an infrastructure for Xen, perfctr-xen, that allows access to the hardware performance counters in a virtual environment. The infrastructure relies on a hypervisor component along with modifications to the guest OS kernel.

[4] mentions the requirement of synchronous interrupt delivery, which should be straightforward for all hypervisors: each performance counter interrupt is generated locally on the PCPU where the VCPU currently resides, and the hypervisor must have a fault injection mechanism that can be leveraged for interrupt injection. [10] describes whole-system profiling with the help of agents at each level of the virtualization hierarchy.

12 Conclusion

This paper presented a proposed solution for virtualizing and time-sharing performance counters while providing reasonable and intuitive counter results. The proposed implementation distinguishes between speculative events, which are allowed to count while a hypervisor is emulating guest code, and non-speculative events, which are faithfully emulated to match in-order program flow. In this way, unmodified profiling tools can be used inside a VM to gather useful performance profiles that do not conceal overheads caused by virtualization.

Acknowledgements. The authors would like to thank Hussam Mousa of Intel for collaboration and investigation of performance counter implementation and usage across processor generations and profiler versions.

References

1. AMD Bios and Kernel Developer Guide, `http://support.amd.com/us/Processor_TechDocs/31116.pdf`
2. Bhargava, R., Serebrin, B., Spadini, F., Manne, S.: Accelerating two-dimensional page walks for virtualized systems. ASPLOS XIII, Seattle, WA
3. CodeAnalyst, `http://developer.amd.com/tools/codeanalyst/pages/default.aspx`
4. Du, J., Sehrawat, N., Zwaenepoel, W.: Performance Profiling in a Virtualized Environment. In: Hotcloud 2010, Boston, MA (2010)
5. Du, J., Sehrawat, N., Zwaenepoel, W.: Performance Profiling of Virtual Machines. In: Virtual Execution Environments 2011, Newport Beach, CA (2011)
6. Gleixner, T.: Linux Performance Counter announcement, `http://lkml.org/lkml/2008/12/4/401`

7. Hauswirth, M., Sweeney, P., Diwan, A., Hind, M.: Vertical Profiling: Understanding the Behavior of Object-Oriented Applications. In: OOPSLA 2004, Vancover, British Columbia, Canada (2004)
8. Intel Software Developer Manual, vol. 3, http://www.intel.com/Assets/PDF/manual/325384.pdf
9. Menon, A., Santos, J.R., Turner, Y., Janakiraman, G., Zwaenepoel, W.: Diagnosing Performance Overheads in the Xen Virtual Machine Environment. In: Virtual Execution Environments, Chicago, IL (2005)
10. Mousa, H., Doshi, K., Sherwood, T., Ould-Ahmed-Vall, E.: VrtProf: Vertical Profiling for System Virtualization. Hawaii International Conference on System Sciences, Koloa, HI
11. Nikolaev, R., Back, G.: Perfctr-Xen: A Framework for Performance Counter Virtualization. Virtual Execution Environments, Newport Beach, CA (2011)
12. Popek, G., Goldberg, R.: Formal Requirements For Virtualizable Third Generation Architectures. Commun. ACM 17(7), 412–421 (1974)
13. VTune, http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/
14. VMware: Timekeeping In Virtual Machines, http://www.vmware.com/files/pdf/Timekeeping-In-VirtualMachines.pdf
15. VMware: VMkperf for VMware ESX 5.0 (2011)
16. Zhang, Y.: Enhanced perf to collect KVM guest os statistics from host side, http://lwn.net/Articles/378778/