

Hardware Support for Cross-Layer PMU Arbitration

Rob Knauerhase[†], Paul Brett[†], Peggy Ireland[‡]
 knauer@jf.intel.com, {paul.brett, peggy.irelan}@intel.com
[†]Intel Labs, [‡]Intel Software and Services Group
 Intel Corporation

Abstract—¹ Intel processors offer PerfMon, a set of hardware events and counters that may be programmed in a number of ways for a variety of uses. Traditionally used for application optimization, we are seeing novel nascent uses throughout the software stack: in operating systems, virtualization hypervisors, and even BIOS firmware.

Conflict for these counters has already been observed, and is likely to worsen. We posit the need for hardware features to allow “reservation” of and exclusive access to hardware counters, and describe a prototype system² to solve the problem.

I. INTRODUCTION AND MOTIVATION

A. A Brief History

The origins of Intel[®] hardware performance monitoring stem from a collection of largely undocumented model-specific performance monitoring features built into the processor, principally so that hardware engineers could validate the design of their functional units. Over time, these model specific features have been exposed to the software community, allowing programmers to use them for application tuning and optimization. In recent processors, Intel has begun to standardize Architectural PerfMon [2] capabilities, providing a consistent set of features both in terms of capabilities and hardware interface. Because Architectural PerfMon is a documented feature, it will be compatibly present on future processors, making its use yet more attractive to software developers.

B. Conflicting uses

Existing uses of PerfMon are largely performance monitoring and debugging; for example, in tools like oprofile and VTune[™] which provide the ability to inspect the behavior of individual applications or whole systems. Other tools (e.g. Tivoli) use PerfMon to describe system utilization in addition to other metrics in software or peripherals.

New uses, however, have emerged, for example recent commercial servers have begun to use PerfMon in the BIOS to do low-level power management; indeed, several highly-ranked SPECpower ([?]) systems achieve their results [?] through this technique. Moreover, the research community has begun to use PMU hardware for observation-based scheduling in the operating system [4] as well as in virtualization hypervisors

[?]. Other current and upcoming research involves using counters for datacenters (chargeback, SLA enforcement) and within runtime environments (dynamic profile-guided optimization). Lastly, new schemes for scheduling tasks[5] and VMs[?],[6] across heterogeneous multicore processors rely heavily on dynamic monitoring via PerfMon.

Historic use of PerfMon assumed a single system-wide privileged entity who could program and reprogram the counters at whim. As more and different uses become prevalent, the problem of conflicting use of the counters is growing. For example, if software entity *A* is using counter IA32_PMC0 for whatever reason, another software entity *B* can, at any time, reprogram that counter for its own use. Not only does this change affect *A*'s behavior, but entity *A* also has no way of knowing that the counter it is using has been programmed for a completely different event.

Further complications arise when migrating among cores in a multi-core system; since counters may be thread-specific, core-specific or shared among several cores, *A* and *B* can co-exist using IA32_PMC0, but only as long as they are never scheduled on the same core. This requirement is burdensome at best, and programmatically inexpressible at worst.

While there have been attempts to develop conventions for counter usage [3], there is no requirement for programmers (especially, obviously, legacy code) to follow the conventions, and no enforcement mechanism for programs which do not behave. In order to ameliorate this “counter chaos”, we propose *Platform Counter Reservation* (PCR), a set of hardware mechanisms to guarantee exclusive use of a counter, and to enable software schemes for reservation and allocation of Architectural PerfMon resources.

In the next section, we describe the hardware “building blocks” and the policies they enable. We then describe our prototype of PCR, and lessons learned from its implementation.

II. PLATFORM COUNTER RESERVATION

A. Environmental Requirements

Among the requirements for PCR's design and implementation were:

- acknowledgement that the end user (or his IT administrator) is in ultimate control of PerfMon resource allocation: the user can run whatever counter-consuming software he wants. Specification of this desire may be made in BIOS setup, from an operating system utility, or elsewhere. PCR hardware features will support this initial setup.

¹NOTE TO REVIEWERS: this is an abridged tech report; many citations (which appear as [?]) are not complete in this version, but will be as part of workshop submission, should our work be accepted.

²This paper describes *research*, and does not describe or promise features in Intel's processor roadmap.

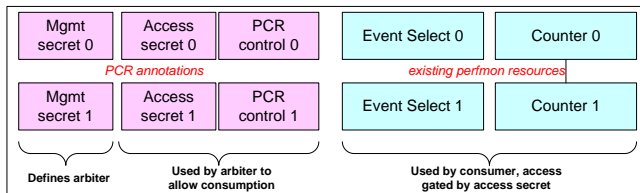


Fig. 1. PCR counter metadata

- compliance with ongoing ecosystem standards and conventions for preventing denial of counter resources to certain entities whenever possible
- compatibility with legacy software: PCR must not break legacy software when counters are not reserved. When counters are reserved, the behavior of legacy software is configurable (so that system software can detect access and trap if need be, etc.)
- simplicity of use: the hardware features must be easily comprehensible to system-software programmers, and software APIs that use PCR should be easily explainable to application developers

B. Hardware Mechanisms

1) *Counter annotations*: Our system annotates each PerfMon counter with metadata. We do not prescribe specific hardware implementations, such as where and how the annotations are stored within a processor core, since different choices vary widely in complexity, implementation cost, validation cost, and so forth.

PCR describes two “cookies”, (see figure 1) which are shared secrets between a software entity and the processor. One of the cookies is for management of the counter (e.g. possession of this secret allows arbitration among counter users). The other is for simple access (e.g. possession of this secret allows programming the counter and consuming counter data). Software may share these cookies among different sub-components as appropriate.

PCR also includes counter annotations for “reservation control”; these values affect hardware behavior when software configures them among several choices. The initial values of these bits are chosen carefully to provide the most basic environment in which PCR can operate; software can select PCR control options according to its level of sophistication. Among the PCR control bits are:

- *mode bits* which control the validity of a management or access cookie in different Intel[®] Architecture processor modes. For example, keys may be specified to be valid only in system management mode (SMM), or only in VMX root mode, or rings 0 and 3, etc. This provides some amount of protection among software layers, as well as making room for new PCR features in the future.
- *error bits* which change the behavior of PCR when an incorrect cookie is presented by software. PCR supports either a “drop” mode (in which reads return a fixed value, and writes are silently dropped), or a “fault” mode in which case a processor fault occurs when software presents a bad cookie. With these options, the system

can be compatible with legacy software (which may not behave correctly, but will not cause a blue-screen or kernel panic), or may include fault handlers for legacy code or other deployment choices.

- *compatibility bits* which describe whether or not to honor existing PerfMon controls in cases where the control affects a number of counters or is stored separately from the performance counter. These compatibility flags allow PCR protected counters to be isolated from global behavior when required.

2) *ISA accommodation of counter usage*: PerfMon is currently exposed via IA32 RDMSR, WRMSR, and RDPMSR instructions, which read and write model-specific registers. Our prototype work explored the trade-offs between:

- adding new PerfMon instructions — by extending the Intel[®] Architecture instruction-set architecture (ISA), we would create opcodes to set cookie values and to provide cookies allowing access to counters. This would allow a “clean slate” implementation, but would also require additional counters or complex methods of mapping the new instructions into existing counters and registers. Adding instructions to Intel[®] Architecture is a decidedly nontrivial effort.
- folding PCR into existing PerfMon — for this option, we created an “alias” table in order to use a separate portion of MSR namespace to denote PCR-compatible vs. legacy counters. For legacy counter access, the processor would assume a reserved value (PCR_DEFAULT_COOKIE).

C. Software Features

1) *Counter arbitration*: PCR, as mentioned, provides “building blocks” on which software can build various policies. PCR hardware modifications were carefully crafted to enable management of counters by one or more management entities, regardless of whether they communicate among themselves. This allows various ecosystems (Linux, Windows, etc.) to develop simple or complex management schemes, and requires only per-ecosystem software APIs, rather than necessitating horizontal standardization between different camps.

A counter arbitrator, regardless of scheme, would start by obtaining management rights to a set of counters. Software desiring to use counters would request access from the arbitrator through arbitrary APIs. The hardware specifically does not restrict choices among APIs or protocols to request and grant counter access in software.

We envision a number of possible software stacks:

- *single master arbiter* — perhaps the simplest case, in which there is a single arbiter which manages every counter resource in the system. Such an arbiter would claim all counters and hold them for the life of the system. There would have to be APIs allowing access to the arbiter (for requesting and relinquishing exclusive access to reserved counters) from all layers of the stack.
- *multiple independent arbiters* — each layer of the stack optionally provides an arbiter for entities in its layer. Upon boot, for example, the BIOS arbiter could obtain one or several counters, leaving the remainder for the

hypervisor or the operating system. Each layer would develop independent APIs and protocols for granting and revoking counter resources.

- *multiple interacting arbiters* — this is similar to the above, except that each arbiter would be explicitly aware of arbiters in other layers. This provides the greatest flexibility, at the cost of greatest complexity. When an arbiter X receives a counter request from a software entity in its layer, it attempts to provide a sufficient counter resource. If the arbiter has no free counters, it can then request resources from other arbiters in the system, allowing dynamic cross-layer arbitration and sophisticated policies for when to grant and revoke access among layers.

III. PROTOTYPE IMPLEMENTATION

A. Platform Prototype

The PCR prototype was built using the Xen[1] hypervisor to implement the modified instructions and additional registers we required. The prototype enabled us to evaluate key design features including the addition of new instructions.

The prototype implemented the following features:

- New RDMSR and WRMSR for access to the lockable counters
- Extended semantics for RDMSR and WRMSR to enable access to lockable counters
- Legacy access to counters using RDMSR and WRMSR
- Locking of performance counters and event selection registers for PERFCTR[0-1], EVNTSEL[0-1]
- Emulation of PEBS_ENABLE and GLOBAL_CTRL enable bits

However, the limitations of virtualization prevented us from fully evaluating performance and cross-layer characteristics of PCR. For time reasons, we did not implement locking of fixed-function counters, or the user-mode RDPMC instruction.

B. OS Prototype

Linux on Intel[®] Architecture already contains 4 kernel operations (`reserve_perfctl_nmi()`, `release_perfctl_nmi()`, `reserve_evntsel_nmi()`, `release_evntsel_nmi()`) to “reserve” and “release” PerfMon registers. Reservation returns success if the specified counter was not already in use; release always succeeds. These operations are exported to kernel-module space to allow external software to acquire the counters and notify other software that they are in use. These functions have no concept of tracking ownership (which entity reserved the counter), and avoid race conditions only by convention. They also have no enforcement capability against bad or buggy software, nor is this “reservation” exposed to layers outside the kernel.

We built three simple Linux arbiters to evaluate operating system enabling complexity, to validate PCR compatibility, and to provide a platform on which to run real software like Intel[®] VTune.

```
void pcr_update_lock(unsigned int counter) {
    if(test_bit(counter, perfctr_nmi_owner) &&
        test_bit(counter, evntsel_nmi_owner)) {
        // unlock counter
        pcr_wrmsr_safe(MSR_IA32_LMSR_AK0+counter,
                      PCR_DEFAULT_COOKIE, secret);
    } else {
        // lock counter
        pcr_wrmsr_safe(MSR_IA32_LMSR_AK0+counter,
                      secret, secret);
    }
}
```

Fig. 2. Minimal Enforcing Master Arbiter implementation in Linux

1) *Minimal master arbiter*: The minimal arbiter maintains the existing Linux kernel interfaces described above for counter reservation, while enforcing the separation of counters between BIOS and OS uses. At boot time, the system executes initialization code which inspects the platform, discovering lockable counters using CPUID, testing the availability of counters by attempting to set a locking cookie, and marking any counters reserved by the BIOS as unavailable to the OS. Remaining counters are reserved for operating system use and made unavailable to the BIOS using the ring control bits in the reservation control register.

Since PCR does not provide a strong security mechanism within a single layer, the arbiter’s locking cookie is stored in a simple static file scoped variable. A more security-focused arbiter could choose between any one of a number of existing techniques to reduce the visibility of the shared secret.

The access keys for the OS accessible counters are set to PCR_DEFAULT_COOKIE to allow legacy client access from within the OS. In order to maintain compatibility with the existing Linux API, performance counters in our prototype are only considered lockable if they can be reserved by the arbiter on every CPU in the system. This simplification eliminates the need to track counter usage on a per-CPU basis.

2) *Minimal enforcing master arbiter*: The minimal enforcing arbiter extends the above interfaces, maintaining the existing kernel interface while enforcing the current implicit kernel counter lifecycle for all existing kernel counter users. Access to the counters is forbidden by PCR if the counter has not been reserved, or after a reserved counter has been released.

Initialization is performed using the same function as the minimal arbiter, but the access cookie is set by default to the secret value rather than to PCR_DEFAULT_COOKIE. The Linux API functions are then modified to call `pcr_update_lock` (see figure 2) upon completion to update the PCR access cookies to reflect the state of the Linux reservation bits.

3) *Fully-enforcing master arbiter*: The enforcing arbiter extends the prototype further by providing, in addition to the legacy interface, a new kernel API which provides acquisition and release of private access to a performance counter by a kernel-mode user. Arbiter initialization and legacy acquisition and release of the counters are performed as per the minimal-enforcing arbiter.

C. Evaluation

Our prototyping efforts enabled preliminary evaluation of a number of aspects of PCR. First and foremost, the stock Linux kernel worked without modification in the presence of emulated PCR hardware. Another major concern was the level of difficulty for software to effectively use PCR features. While our prototype was not product-quality code, we were able to implement each of the three Linux arbiters in less than 200 ESLOC (executable source lines of code). This provided confidence not only that our hardware features were sufficient for the task, but also that enabling of the kernel to use those features for arbitration was not beyond the reach of typical OS developers.

The nature of the Xen prototype did not provide a good basis for performance analysis (virtualization overhead vastly dominated additional processing time for key management and use), but we were able to use the arbiters with other internal (proprietary) prototyping tools to determine that our additions to RDMSR and WRMSR would entail a small delta in performance, significantly less than the benefits gained by our related research into effective use of the counters for scheduling and migrating tasks in the OS and hypervisor.

IV. FUTURE WORK

There are a number of questions we have not yet addressed. Both new-ISA and compatible-ISA options for PCR provide opportunities for counter aliasing. With such aliasing, a task or VM that uses a particular counter would not be precluded from migrating to a core on which that counter is already reserved; the system software could reserve another counter and simply re-map the alias table appropriately.

Independent of aliasing, system software could reserve one or more counter instances on multiple cores, and intelligently schedule tasks or VMs based on its knowledge of who expects which counters to be reserved. In this manner, migration could happen among a number of cores, as long as no two entities with similar counter expectations were mapped to the same core simultaneously.

Security was a concern for PCR design. In the absence of particular security hardware (e.g. a trusted platform module (TPM) or specially-encrypted storage), we were unable to provide strong guarantees against malicious software. We rationalized this concern by noting that any software in a position to attack PCR would also have other softer targets that would be much more easily exploited than PCR. More detailed security analysis remains for future effort.

We considered the tradeoffs for various sizes (bit-widths) of management and access cookies, but our conclusions were highly processor-specific. Our prototype used a 48-bit cookie, which not only fit nicely within existing model-specific registers, but also provided some measure of “security” (not cryptographic security) in that exhaustive searches of the cookie space would probabilistically be sufficiently time-consuming to be a detriment against “cracking” cookies. Determination of optimal key sizes (factoring in other aspects of processor architecture) has not been addressed by our work so far.

Most importantly, while PCR solves problems related to counter contention, it does not provide policies to determine

which entities/layers can reserve what number of counters. Indeed, PCR as we describe above would enable, for example, a pathological system BIOS to reserve all PerfMon resources, effectively starving higher layers. Much of this problem is outside the scope of PCR, but we continue to investigate hardware features that would help prevent greedy software entities from completely locking out other PerfMon consumers.

Lastly, while we are encouraged that PCR provides an easily-accessible and low-impact set of features, we do recognize that adoption of any such scheme would require both evangelism among ecosystem players (BIOS writers, virtualization vendors, operating-system developers, etc.), as well as integration with their code and release/upgrade schedules. Until and unless PCR-style functionality is a committed feature on processor roadmaps, these problems remain academic.

V. CONCLUSIONS

We believe that trends of PerfMon usage among new and varied players will lead to “counter chaos”, as independent software entities compete for limited PerfMon resources in the absence of either protection from or notification about conflicting usage. As Architectural PerfMon matures, we expect the trend to accelerate as developers feel more comfortable with cross-generation and cross-processor-family support for PerfMon hardware.

We have begun research into mechanisms that will allow reservation of counter resources by software arbitrators, and guaranteed access to those resource by grant from an arbiter. Our exploration has resulted in a small set of hardware features which enable software to “count on the counters”. We have developed initial prototypes both to determine the utility of our features, and to explore various use models for software entities from BIOS to applications.

Our research is encouraging, in terms of utility, performance, and implementation complexity. We will continue investigation along these lines, with appropriate interaction and input from developers, in hopes of advancing the functionality of Intel’s microprocessors for advanced performance monitoring and advanced new uses of monitoring data throughout the system.

REFERENCES

- [1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, New York, NY, USA, Oct. 2003. ACM.
- [2] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B: System Programming Guide, Part 2*. Intel Corporation, 2009.
- [3] P. Ireland and S. Kuo. Performance monitoring unit guidelines. <http://software.intel.com/en-us/articles/performance-monitoring-unit-guidelines/>.
- [4] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS observations to improve performance in multi-core systems. *IEEE Micro*, 28(3):54–66, May 2008.
- [5] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the Fifth European conference on Computer Systems*, New York, NY, USA, Apr. 2010. ACM.
- [6] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *Proceedings of the Fifth European conference on Computer Systems*, pages 139–152, New York, NY, USA, Apr. 2010. ACM.