Cache QoS: From Concept to Reality in the Intel® Xeon® Processor E5-2600 v3 Product Family

Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, Ravi Iyer

Contact: andrew.j.herdrich@intel.com, ravishankar.iyer@intel.com

Intel Corporation

Abstract

Over the last decade, addressing quality of service (QoS) in multi-core server platforms has been growing research topic. QoS techniques have been proposed to address the shared resource contention between co-running applications or virtual machines in servers and thereby provide better isolation, performance determinism and potentially improve overall throughput. One of the most important shared resources is cache space. Most proposals for addressing shared cache contention are based on simulations and analysis and no commercial platforms were available that integrated such techniques and provided a practical solution. In this paper, we will present the first set of shared cache QoS techniques designed and implemented in state-of-the-art commercial servers (the Intel® Xeon® processor E5-2600 v3 product family). We will describe two key technologies: (i) Cache Monitoring Technology (CMT) to enable monitoring of shared cache usage by different applications and (ii) Cache Allocation Technology (CAT) which enables redistribution of shared cache space between applications to address contention. This is the first paper to describing these techniques as they moved from concept to reality, starting from early research to product implementation. We will also present case studies highlighting the value of these techniques using example scenarios of multi-programmed workloads, virtualized platforms in datacenters and communications platforms. Finally, we will describe initial software infrastructure and enabling for industry practitioners and researchers to take advantage of these technologies for their QoS needs.

1. Introduction

Over a decade ago, multi-core (CMP) platforms started to become the norm for providing increased performance and throughput for high performance server systems. Around the same time, a few researchers [2, 10, 13] identified that multi-core platforms will face quality of service challenges due to contention between shared resources. These shared resources include shared cache space, memory bandwidth and other invisible shared resources in the platform that are not exposed to system software (operating systems, virtual machine monitors, etc.). This contention affects the ability to provide performance determinism and prioritization between different applications co-running on the same platform. As a result, applications running simultaneously on a platform with other workloads could suffer significant variability in observed performance. To address this problem, researchers have analyzed the problem over the last decade and proposed solutions [1, 5, 6, 7, 11, 12, 15, 17, 20, 21, 22, 23, 24, 27] to effectively partition the resources across multiple applications. A large number of these papers focused on the shared cache contention as a key problem since it carries a first-order impact to performance determinism and quality of service. In this paper we focus on shared cache contention as well and show the path from concept to productization of two important cache QoS techniques.

The Cache QoS techniques demonstrated in this paper cover two important problems: (a) the ability to monitor and profile the shared cache space usage of applications when running together and (b) the ability to enforce and allocate shared cache space usage for applications based on OS/VMM hints and policies. The two techniques -- Cache Monitoring Technology (CMT) and Cache Allocation Technology (CAT) respectively – have been developed from concept to productization and are now launched as part of the Intel® Xeon® processor E5-2600 v3 Family (formerly codenamed *Haswell*), with many usages in server and communications platforms. Such server platforms are widely used for high performance computing and cloud datacenters [32, 33] for instance. In this paper, we will first describe the early research that formed the motivation and basis for these techniques and then describe the additional architectural and productization changes and learnings that were required to implement and deploy the features as part of the Intel Xeon processor family. In particular, we will also describe the enumeration, scalability and flexibility that Intel's CMT and CAT technologies offer by providing Resource Monitoring IDs (RMIDs) and Classes of Service (CLOS) as key abstractions in the architectural interface.

Based on the Intel® Xeon® Processor E5-2600 v3 product family we will show measurement-based platform experiments demonstrating benefits and trade-offs of employing CMT and CAT. The case studies include multi-programmed workloads as well as communication-centric virtualized workloads to show the range of QoS needs. Using CMT and CAT, we demonstrate how the hardware techniques coupled with software policies can be effective in managing resources efficiently and improving isolation, performance determinism and even overall throughput in some cases. Along the way, we also demonstrate practical usages of RMIDs and CLOS.

978-1-4673-9211-2/16/$31.00 ©2016 IEEE

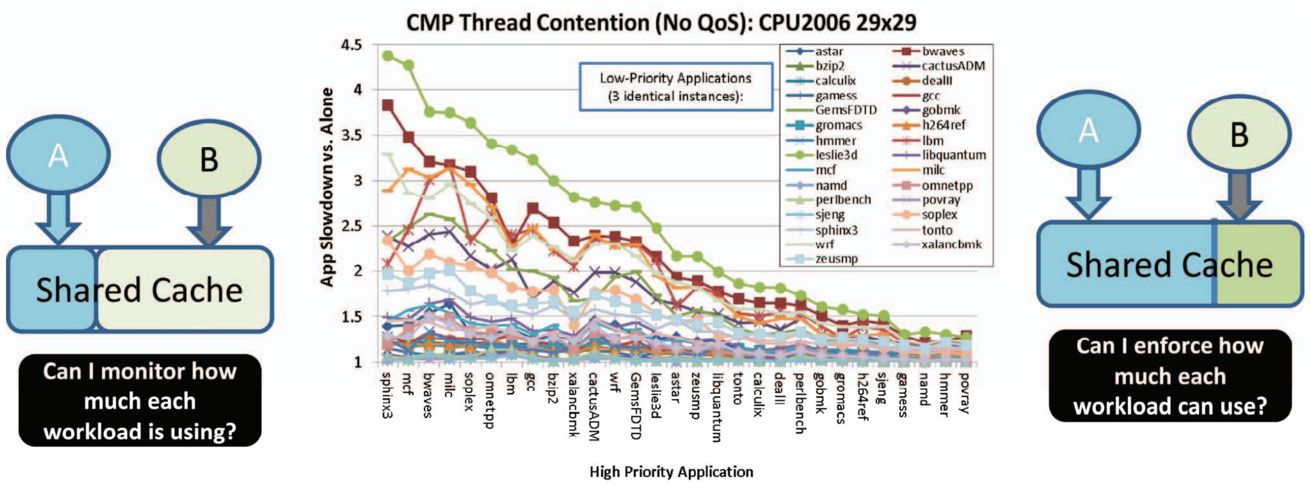


Figure 1: QoS Challenges and Implications due to Invisible Shared Resources

2. Motivation and Background

In this section, we motivate the need for Cache QoS and present key considerations for designing a usable solution.

2.1 QoS Challenges & Research Overview

When multiple workloads run simultaneously on a multi-core platform, the impact of their resource contention depends significantly on the resource consumption of each workload. *Figure 1* illustrates the effect of resource contention on an Intel Xeon processor-based platform using SPEC* CPU 2006 workloads for demonstration. The figure shows the slowdown that a (high priority) workload (listed on the x-axis) suffers when running simultaneously with three copies of a (low priority) workload (shown on the legend). This execution time slowdown is compared to the scenario where the workload is running alone. As can be observed from the figure, the slowdown ranges from 1.0x to 4.4x depending on the mix of workloads selected to run simultaneously. It should be noted that each of these workloads are running on a different core (thus no CPU time contention) but they share cache space and memory bandwidth resources and thereby suffer from a slowdown. Control over these shared resources is currently exposed to the system software and therefore contention for these resources cannot be controlled today. This observation leads to two key questions that need to be answered when running multiple applications simultaneously on a platform:

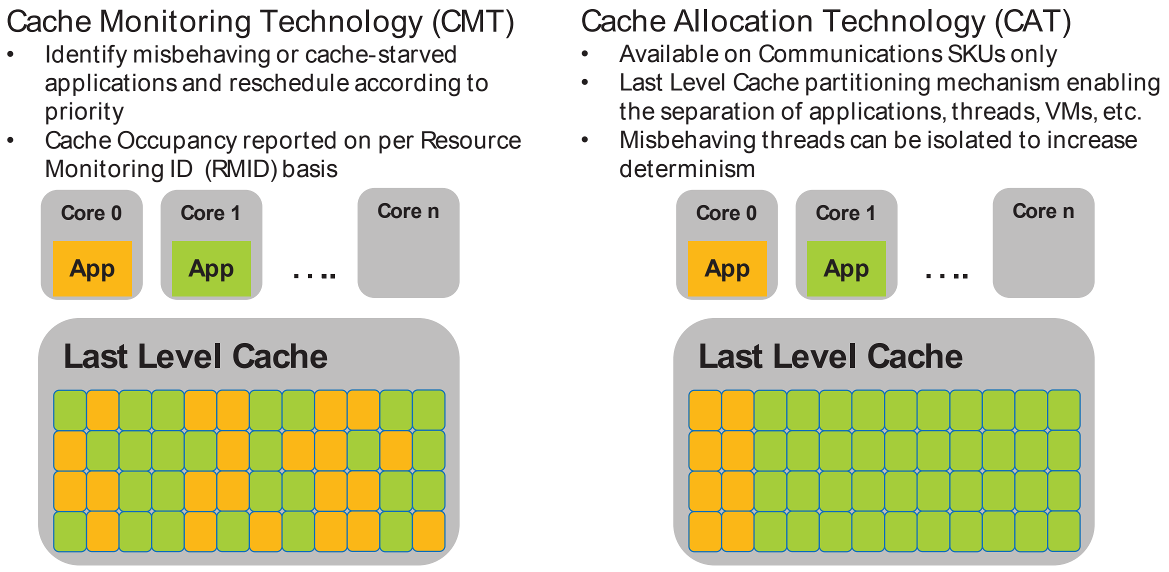
- **Visibility into resource usage:** *Can the platform enable visibility into the resource usage for each of the workloads running on it? Figure 1* illustrates this with two workloads (A & B) and the goal is to understand how much capacity A is using in shared cache (for instance) and how much B is using in shared cache.
- **Ability to enforce resource usage:** *Can the platform allow guided redistribution of resource usage for each of the workloads running on it? Figure 1* illustrates this with two workloads (A & B) and the goal is to enable the ability to specify how much capacity A and B can each use in shared cache (for instance).

In the coming sections we will describe Cache QoS technologies (Cache Monitoring Technology and Cache Allocation Technology). The goal of these technologies is to establish an architecture that addresses the two questions above and provides system software the ability to monitor and enforce shared resource usage. However, before jumping into implementing such capabilities, it is important to highlight productization considerations as described in the next subsection.

2.2 Key Considerations for Productization

When developing new capabilities and architectural interfaces exposed to systems software, it is important to keep the following productization considerations in mind:

- **Usage Flexibility:** There are many different usages for such capabilities ranging from addressing isolation needs (e.g. communications platforms running real-time tasks simultaneously with non-real-time workloads) to addressing prioritization needs (e.g. datacenter platforms running multiple virtual machines that have different SLAs) to addressing resource efficiency (e.g. HPC machines trying to achieve highest performance or throughput by optimizing resource usage). The HW capabilities exposed should allow systems software to make policy decisions according to the usage.
- **Architecture Consistency:** The underlying HW/SW interface and architecture should ideally be invariant to (a) changes in platform configurations (e.g., number of sockets, number of cores, different cache sizes), (b) software-friendly in terms of ease of use and latency of register access and context/save restore and (c) tolerant to future changes in cache/memory hierarchy or architecture. In addition, the software interface exposed should be similar for different types of resources in different types of platforms to enable ease of use and avoid need for customization as much as possible. In addition to architecture consistency, it is also important to keep the implementation light-weight (low cost in area and power) and flexible (for future scalability).


Cache Monitoring Technology (CMT)
Identify misbehaving or cache-starved applications and reschedule according to priority
Cache Occupancy reported on per Resource Monitoring ID (RMID) basis
Core 0
App
Core 1
App
....
Core n
Last Level Cache
Cache Allocation Technology (CAT)
Available on Communications SKUs only
Last Level Cache partitioning mechanism enabling the separation of applications, threads, VMs, etc.
Misbehaving threads can be isolated to increase determinism
Core 0
App
Core 1
App
....
Core n
Last Level Cache

(a) Cache Monitoring Technology Example *(b) Cache Allocation Technology Example*

Figure 2. Overall QoS Framework and Resource Monitoring/Allocation Flows

- **Software Support:** A new architecture/interface typically requires a significant amount of enabling for various OSes, VMMs, tools, etc. As a result, implementing reference solutions that allow for ease of deployment is highly desirable for commercial and academic use. This includes enabling open-source software support (e.g., Linux*) for wide access to the features.

When developing Cache Monitoring Technology and Cache Allocation Technology all the above considerations were carefully addressed. In the next section, we detail the Intel CAT/CMT technologies and describe the HW/SW interface and architecture/capability in more detail.

3. From Concept to Reality: Addressing QoS

The early research behind QoS technologies was first published in ICS 2004 [10] (cache allocation) and PACT 2007 [30] (cache monitoring) respectively. *Figure 2* (top) shows how these technologies work together with system software policies to form a holistic QoS framework. In the early works, the main concepts introduced were the use of classes of service (CLOS) to prioritize resource allocation, and the use of resource monitoring IDs (RMIDs) to enable visibility into resource usage. However, converting these concepts into real product solutions required careful separation of the architectural interface and implementation to provide flexibility and also careful consideration of productization aspects described in the previous section. We now describe the resulting QoS technologies deployed in the Intel Xeon Processor E5-2600 v3 family.

3.1 Cache Monitoring Technology (CMT)

Intel's Cache Monitoring Technology (CMT) provides monitoring of shared L3 cache (specifically cache occupancy) in real time at granular level of detail for all RMIDs simultaneously. In order to provide maximum monitoring flexibility, CMT operates independently of other technologies such as performance monitoring counters or virtualization technologies and enables continuous monitoring of shared cache occupancy based on a pool of monitoring IDs made available by the platform (RMIDs). The key value proposition and usages for CMT were described in early work [30] as follows:

- **Performance profiling and characterization –** While running with others, how much cache space does an application consume and how is performance affected?
- **OS/VMM Scheduling for Perf Management:** In a multi-node platform running multiple applications, on which core/cache would an application run best?
- **Metering for Chargeback or Resource Limiting:** How should an infrastructure be built to charge customers for resource use, or limit resource use to a running average limit?
- **Quality of Service and Efficiency:** When combined with Cache Allocation Technology (described later), how can we share resources between multiple applications to improve both QoS and overall throughput/efficiency?

3.1.1 CMT Usage Flows

Figure 2a illustrates the basic usage model intended for CMT. After enumerating the presence and capabilities of the technology using CPUID [8], the OS/VMM can then select which threads/apps/VMs should be monitored, then periodically read back the occupancy data as required. Monitoring of cache occupancy is accomplished via the introduction of an intermediate layer of abstraction: a pool of *Resource Monitoring IDs* (RMIDs). At runtime, an enabled OS or VMM can associate an arbitrary mix of threads, applications or VMs with a given RMID based on system monitoring needs. The platform hardware maintains occupancy counters per RMID continuously and these occupancy values can be read back later on a per-RMID basis. An illustration of this association is shown below in Figure 3.

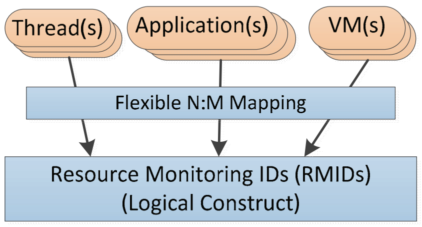


Figure 3. RMID Mapping: Any combination of one or more threads, apps or VMs can be associated with RMID

3.1.2 CPUID Enumeration

The *CPUID* instruction is used to enumerate details of the CMT feature, including parameters that may change over time, such as how many RMIDs are supported on the platform. Details of the CPUID enumeration process are discussed in the Intel Software Developer's Manual [8, 9].

3.1.3 RMID Association

Architecturally, a register is added per logical thread which associates an RMID with the running software thread (*IA32_PQR_ASSOC*), also known as the "PQR" register. When an application is scheduled onto a core the OS or VMM updates the PQR with the RMID of the thread being swapped on (*Figure 4)*, allowing the hardware to track the resource utilization of the given thread via its associated RMID. The definition of the PQR register is provided in a later subsection in *Figure 6*.

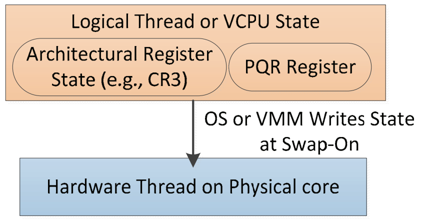

Logical Thread or VCPU State
Architectural Register State (e.g., CR3)
PQR Register
OS or VMM Writes State at Swap-On
Hardware Thread on Physical core

Figure 4. The PQR register is added to the logical thread or VCPU state and is written at context swap-on time

3.1.4 L3 Cache Occupancy Reporting

Through the process of associating the threads or virtual CPUs (VCPUs) to be monitored with RMIDs and updating these at context swap time, accurate measurements of L3 cache occupancy or other events can be automatically collected by the hardware and values can be read back periodically by software. Software can then use the values directly for fine-grained monitoring, or aggregate and store the results with cache occupancy as the independent variable (for instance, when the application consumed 22-23MB of cache, its performance level averaged to a certain value). In order to report results back to software, two registers are provided. The first, *IA32_QM_EVTSEL*, allows software to specify an {*RMID*, *Event Code*} pair for which data is retrieved. The RMID is previously assigned per thread/app/VM by the OS/VMM or user depending on the use model, and the *Event Code* specifies the type of data to retrieve, for instance L3 cache occupancy.

Once an {*RMID, Event Code*} pair has been specified in the event selection register, hardware looks up the occupancy and provides the result in the *IA32_QM_CTR* MSR, which includes bits for error reporting to ensure that results are valid. The raw counter value retrieved is then multiplied by a conversion factor (upscaling factor, previously retrieved from CPUID) to obtain cache occupancy in bytes. It is recommended that in order to provide accurate and timely occupancy that software poll occupancy values at a frequency of greater than 1Hz. Faster polling rates can capture application behavior across variable phases, while slower polling rates (down to 1Hz for practical purposes) can reduce overhead.

3.1.5 CMT Implementation

A key to successfully productizing cache monitoring technology was to keep the cost of maintaining the tags and the counters extremely inexpensive. In the original monitoring [30] proposal, the implementation entailed tagging a subset of cache lines in the shared L3 with the resource monitoring IDs. This implementation of CMT was already low area overhead (0.125% of cache), but it should be noted that the product implementation takes it a step further and reduces this cost even more by employing optimizations within the microarchitecture and cache implementation. While the details of this scheme are not described here, it should be noted that this was critical to ensure that future implementations of monitoring were scalable, flexible and avoid requiring the introduction of the additional tag space (even though small) as described in the original proposal.

In the first CMT implementation in the Intel Xeon E5 v3 family, two RMIDs are supported per hardware thread. With an 18-core system (36 hardware threads), threads can be tracked with 72 RMIDs. It is possible to implement RMID use and recycling algorithms of several types with a selectable tradeoff between the number of threads that can be tracked and instantaneous accuracy. Examples include:

1. **Number of threads < number of RMIDs:** Assign one RMID per thread. Highest accuracy.
2. **Number of threads > number of RMIDs:** Employ a LRU algorithm amongst the pool of RMIDs. When swapping a thread onto a core, select the least-recently used RMID (and check to ensure that occupancy is close to zero, if not select another). This ensures high accuracy at the cost of low checking overhead.
3. **Number of threads >> number of RMIDs:** Continuous recycling: As a static rule, always pick the LRU RMID from the pool. Since the system is highly

active (many running threads), the occupancy of unscheduled RMIDs will converge to zero quickly and they can be rapidly reused (since half of the total RMIDs are always unscheduled at any given time with number of RMIDs as 2x the hardware threads).

A detailed analysis of RMID swapping impact is provided in a later section on system/software considerations.

3.2 Cache Allocation Technology (CAT)

In the Intel Xeon processor E5 v3 product family a number communications-centric processor models are enhanced with Cache Allocation Technology (CAT), a feature that enables software-guided hints for partitioning of the last-level (L3) cache between running applications. The CAT architecture is separate from, but similar to, the previously described CMT architecture. Similarities include the introduction of an intermediary abstraction called a *Class of Service* (CLOS, *Figure 5*) into which threads, applications or VMs can be grouped in order to provide abstraction between software and hardware cores.

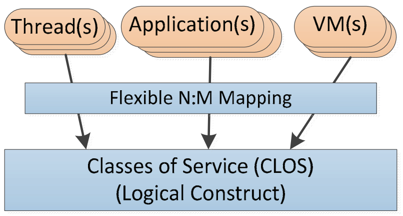


Figure 5. From threads, apps or VMs to Class of Service (CLOS)

The *CLOS* abstraction also shares the same per-thread register (e.g. both RMID and CLOS are specified per logical thread within the same PQR register, with the full definition shown in Figure 6). One key consideration however is that the *CLOS* abstraction is distinct from the *RMID* abstraction, allowing monitoring features to be implemented and scaled independently from allocation technologies (and vice versa).

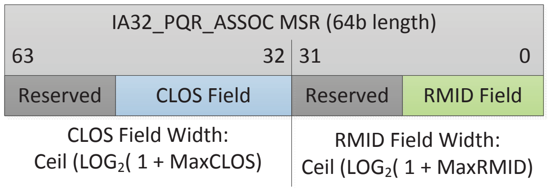
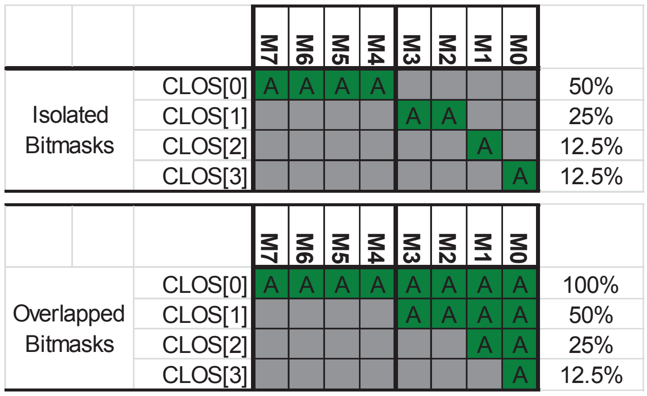

IA32_PQR_ASSOC MSR (64b length)
63
32
31
0
Reserved
CLOS Field
Reserved
RMID Field
CLOS Field Width:
Ceil (LOG$_2$(1 + MaxCLOS)
RMID Field Width:
Ceil (LOG$_2$(1 + MaxRMID)

Figure 6. PQR Register per logical core w/ RMID & CLOS

Enumeration via CPUID is also handled via separate sub-leaves (e.g., details surrounding monitoring features are enumerated in the CPUID.0xF sub-leaf, while allocation details are enumerated in CPUID.0x10). This symmetric yet distinct implementation of monitoring and allocation enables the features to be independently enabled or omitted and scaled based on product needs.

The implementation chosen bears many similarities to the "application priority" approach from [10], which suggests tracking an indicator of thread priority within the process or thread context information maintained by the OS or VMM. In order to provide configurable cache allocation, each CLOS is associated with a capacity bitmask, which specifies the fraction of last-level cache into which members of the CLOS can allocate data (*Figure 7*).



		M7	M6	M5	M4	M3	M2	M1	M0	
Isolated Bitmasks	CLOS[0]	A	A	A	A					50%
	CLOS[1]					A	A			25%
	CLOS[2]							A		12.5%
	CLOS[3]								A	12.5%
		M7	M6	M5	M4	M3	M2	M1	M0	
Overlapped Bitmasks	CLOS[0]	A	A	A	A	A	A	A	A	100%
	CLOS[1]					A	A	A	A	50%
	CLOS[2]							A	A	25%
	CLOS[3]								A	12.5%

Figure 7. CLOS Bitmasks: Isolated and Overlapped

The bitmasks provide a convenient abstraction of cache capacity independent of the underlying cache implementation and independent of the partitioning scheme. In set-associative caches, for instance, the capacity mask may map to a few ways, or in other implementations to a probabilistic allocation mechanism [1]. As shown in *Figure 7* with an example bitmask length of eight bits, the use of capacity bitmasks enables user-specified control over the degree of resource sharing vs. isolation. This in turn enables multiple software performance management schemes, such as those that assign applications into gold/silver/bronze application tiers with differing access to shared resources, or schemes which partition shared resources in a fairness-driven model.

3.2.1 Cache Allocation Usage Flows

The usage flow for Cache Allocation Technology (*Figure 2, right side*) is similar to CMT in that an enumeration stage is required to verify the presence of the feature, then a configuration stage which involves setting up the capacity bitmasks to specify the desired cache allocation scheme (e.g., overlapped masks, isolated, or a mix). Once masks are configured, threads are associated with the CLOS which map 1:1 to the masks enabling CAT. All of these settings may be dynamically updated at runtime, enabling an OS or VMM to adapt to application behavior changes in real-time.

3.2.2 Cache Allocation Enumeration Process

In the initial implementation of CAT in certain Xeon processor E5 2600 v3 models, CAT enumeration is performed in a model-specific fashion, based on the brand string, which encodes the model of the processor, and certain processor models are specified to support the feature. In successive generations it is expected that CPUID will be used for enumeration, enabling application software to adapt to parameters which may change between generations including the number of CLOS available on the platform or the capacity bitmask length (e.g., 8 bits vs. 16 bits).

3.2.3 Cache Allocation Bitmask Configuration

Once the presence of the CAT feature has been confirmed, software may configure the masks to provide isolation, overlap or a combination between the CLOS (*Section 3.2, Figure 7*). Overlapped configurations generally provide better throughput, while isolated masks generally provide better performance isolation and determinism. Note that as the system boots, all masks are initialized to the full length of the capacity mask, meaning that CAT is disabled by default. All threads are initially configured to use CLOS[0].

3.2.4 Thread Association

Once the masks have been configured, threads can be associated into Classes of Service as shown in Figure 8, an example wherein four CLOS are used among the threads, and each CLOS has a preconfigured bit mask. In this case CLOS[0] can allocate anywhere in the cache (green highlighting indicating that the mask bits are all set, and this implies that this CLOS contains high-priority applications). By contrast, CLOS[1] only has a single bit set, and can only allocate into a small portion of the cache, implying that CLOS[1] contains a set of much lower-priority applications.

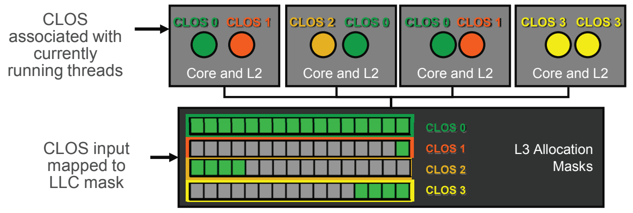


Figure 8. Mapping Threads to Classes of Service to Masks

3.3 System & Software Enabling Considerations

Utilizing the Monitoring and Allocation capabilities is straightforward from a code development perspective with model specific registers (MSRs) providing the interface to set up. All modern Operating Systems provide API's that enable users (with the appropriate privileges) to read and write the MSRs. For example, Linux provides the *msr-tools* package which integrates both the *rdmsr* and *wrmsr* commands. Microsoft* Windows* has a similar interface. There are two approaches to monitoring: the platform could be monitored utilizing a standalone / non-scheduler-based approach, or scheduler-based monitoring. The latter is preferred as it enables easier tracking of applications and/or VM occupancy as they migrate across cores

- **Standalone Cache Monitoring** looks at the Last Level Cache usage from a Core or Thread perspective, regardless of what task is executing. An RMID is statically assigned to a Core/Thread and periodically the occupancy is read back. If the platform has been statically configured and applications have been pinned to resources then this method will yield appropriate results. If system administrators are interested in whether the platform is suitably balanced and there are no misbehaving applications, this approach is acceptable.
- **Scheduler based Cache Monitoring** involves enabling the operating system scheduler. In the previously discussed static assignment / standalone approach the RMID/CLOS do not track the process id and therefore occupancy cannot be reported on a per application basis without pinning. In order to track the applications as they migrate across cores scheduler changes are required. Software (e.g., administrators using monitoring tools) assign an RMID to a process, and in turn the scheduler will associate the core with the appropriate RMID when the application of interest is scheduled to execute on a core or thread. When the application is de-scheduled or migrated to a different core or thread, the scheduler updates the RMID assignment to make sure occupancy is only billed against the correct RMIDs and then only when the application is executing. System software is also responsible for any remapping/re-assignment which may be required across processor sockets. Since RMIDs and CLOS are local per socket, when an application is moved to another socket, the OS/VMM is responsible for finding an available RMID and CLOS on the destination socket to continue to monitor or enforce QoS.

To enable the above, software development initiatives that focus on monitoring implementation are described below.

- **Cache Monitoring and Allocation Library:** This standalone library (available at: https://01.org/) enables developers to monitor the L3 cache occupancy on a per core or thread basis. In addition this utility enables users to set up the Cache Allocation functionality. When the library / application initially comes up it will check for both Cache Monitoring and Cache Allocation support. Once initialization is complete the monitoring functionality provides a "top"-like interface listing the last level cache occupancy on a per core/thread basis. Once system administrators have monitored for period of time the appropriate partitioning can be applied utilizing the Cache Allocation functionality. With both the monitoring and allocation capabilities present in the platform feedback loops can be developed that dynamically change a core / thread allocation to a Class of Service. The library has been included in Ubuntu* 16.04 (Xenial Xerus), and can be installed using the package manager: *sudo apt-get install intel-cmt-cat*
- **Cache Monitoring cgroups implementation:** Linux supports the ability to group certain tasks together using the *cgroup* (control group) subsystem. The CMT monitoring cgroup along with Linux scheduler changes enable the ability to track the RMID assignment as processes or threads are moved between CPUs. The CMT cgroup supports multi-hierarchy groups. A monitoring group accumulates the L3 cache occupancy of all of its

child groups and the tasks directly present in its group. Patches to the Linux kernel and cgroup infrastructure are available here: https://github.com/fyu1/linux/tree/cat16.1

Intel Cache Monitoring Perf Implementation: The Linux *perf* Linux tool provides an interface into kernel-based performance counters. An extension has been developed to support Cache Monitoring. The name of the new event is *intel_qos/cache_occupancy/*. This returns occupancy in bytes. To make sure that the occupancy associated with process ID is accurate the perf kernel component associates the RMID with the running thread only when the PID is scheduled on a CPU. When the Linux scheduler swaps the process the RMID will no longer be associated with the CPU. In addition to RMID tracking, perf also includes process or thread inheritance support. Cache Monitoring support is available mainstream as of Linux kernel v4.1.
The next major consideration for CMT/CAT is the continuous use of finite RMIDs and CLOS. Here we discuss RMID recycling considerations as a case study.

RMID Recycling Considerations: Since RMIDs are a finite hardware resource used to track the occupancy of an application, core or thread, there might be the requirement to recycle RMIDs to keep track of more applications than there are RMIDs available. The following experiment measures the time it takes for occupancy data associated with an RMID to stabilize for an application that streams data through the cache. *Figure 10* displays a last level cache occupancy trace for the duration of the experiment. The platform utilized for this experiment integrates an Intel Xeon processor E5-26xx v3 with Cache Monitoring Technology and CentOS* 6.x Linux kernel. The platform is idle at the start of the trace and an application that reads random data from memory (such as STREAM* benchmark [31]) is affinitized to core 13. At time stamp T(7s) execution starts and consumes all available last level cache, at time T(12s) the application terminates.

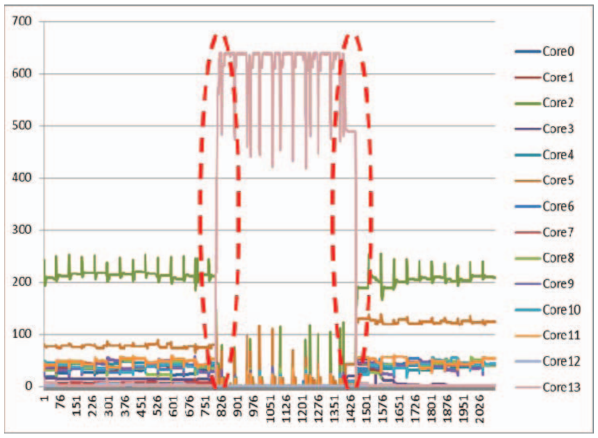


Figure 10. RMID recycling experiments

The occupancy ramp is largely dependent on stride and rate by which the target application reads data from memory. *Figure 11* (top) zooms in on the ramp, and it is clear that the streaming application's occupancy stabilizes after 112ms. At that time most of the code and data present in the last level cache belongs to the streaming application executing on core 13. The drop in the curve is due to another core causing eviction of the code or data associated with the streaming application. *Figure 11* (bottom) zooms in on the rate of decrease, in this example the occupancy remains stable for 488ms then sharply drops to near 0 in 30ms. The reason why the occupancy remains stable for 488ms is due to the fact that the platform is idle (aside from OS tasks) and at the point where the OS schedules one of its tasks like a GUI update thread the occupancy drops as the new thread evicts old data. As discussed earlier, RMIDs are a finite resource and should be managed carefully. The next section presents the effects of swapping RMIDs on a running thread to study the rate by which RMID's converge.

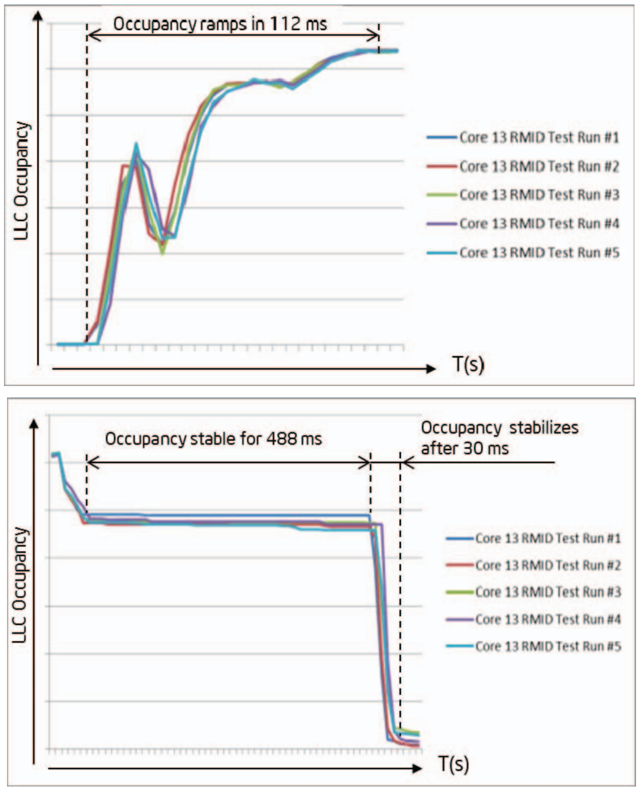
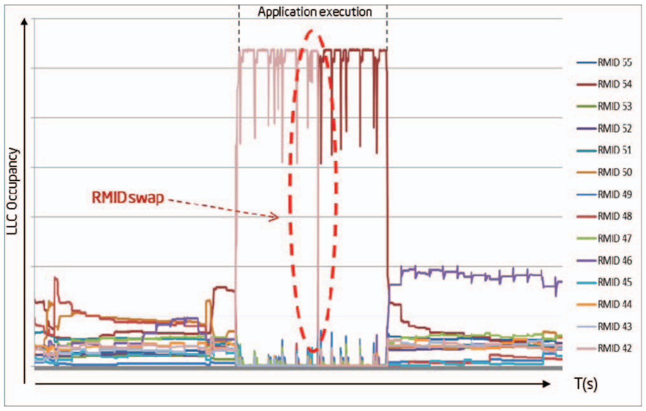


Figure 11. Occupancy Ramp and Stabilization

Figure 12 (top) demonstrates an L3 occupancy trace of the streaming application used in previous experiments. In this example, half way through the execution, the RMIDs are swapped. Due to the streaming nature of the application, RMIDs converge in a short period of time. Figure 12 (bottom) zooms in on the area of interest. The L3 occupancy associated with RMID 42 remains high until the swap, new code and data brought into the L3 after the swap causes a rapid reduction of the L3 occupancy associated with RMID 42. Over 5 samples measured stabilization of the occupancy takes roughly 16ms. In order to re-use RMID

42, a delay of 16ms would be required before associating this RMID with a new application.



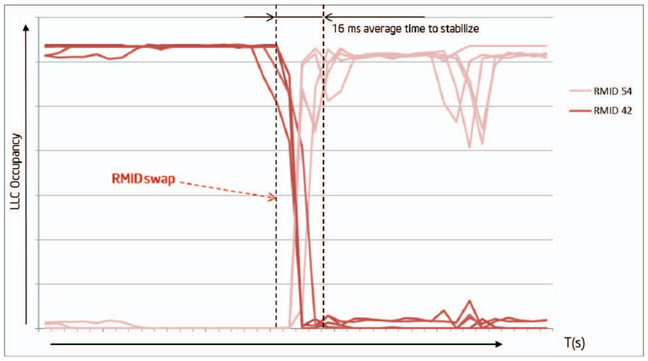


Figure 12. RMID Tracing and Swap Experiment

4. Case Studies on the Intel Xeon processor E5-2600 v3 Family

In this section, real case studies are presented showing the benefits of CMT and CAT. The system was configured with a 2.6GHz processor (pinned at 2.6GHz) with 14 cores, 35MB L3 cache, SMT off, disabled and 64GB of DDR4-2133 memory. Benchmarks and workloads used will be described in subsections below.

4.1 Example Baseline Contention Effects

Baseline contention effects can be demonstrated by picking a sample workload, affinitizing it to one core, and running a series of other applications on the remaining thirteen cores. Data is presented in this section for SPEC CPU2006 as a proxy for HPC workloads (SPEC FP) and since it includes a diverse set of applications. To demonstrate one example here, consider *bzip2*, a widely used compression benchmark as the test application. While running other applications also from SPEC CPU2006 [25] along with bzip2, *Figure 13* shows that *bzip2* performance varies substantially. In the best case, *bzip2* performance is minimally affected by shared cache or memory bandwidth contention and does not slow substantially. In the worst case, however, we observe a slowdown of nearly 5X for the *bzip2* application due to heavy shared cache and memory bandwidth contention. This performance loss and variation is user-visible and can be diagnosed/mitigated by CMT/CAT respectively.

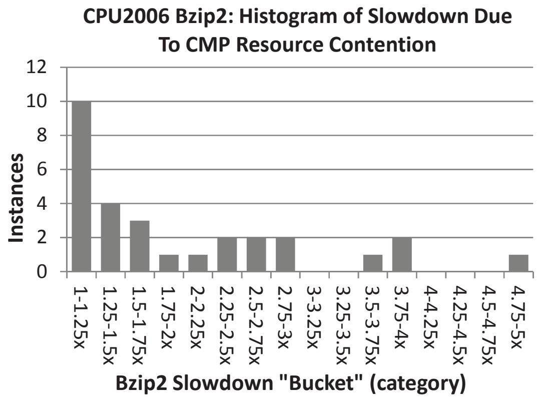


Figure 13. Baseline Contention (bzip2) causing runtime slowdown when running pairwise with 29 applications (CPU2006)

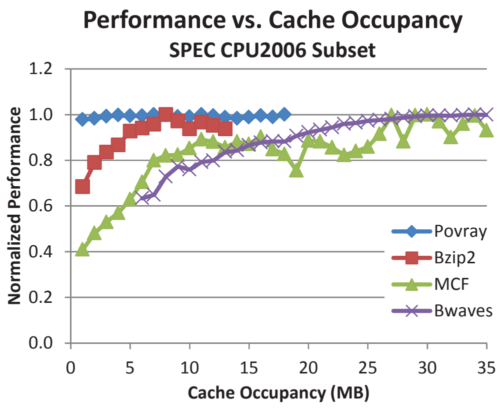


Figure 14. Cache Occupancy & Perf Profiles

4.2 CMT Use Cases and Data

CMT can be used to profile applications over a period of time without needing to have insight into what the underlying application is doing. By sampling over time and aggregating, a workload cache sensitivity curve can be built (*Figure 14*) which correlates application cache occupancy to performance. By tracking occupancy, performance can be predicted and confirmed, and hypothetical questions can be answered (e.g., how much performance would be lost if available cache footprint were to shrink by 5MB). In the example shown in Figure 14, *Povray* (raytracer which is L1-resident) shows little cache sensitivity (no performance variation vs. cache size). Conversely an application like *Bwaves* shows sensitivity over a wide range. As a result, CMT curves can be used to directly infer impact of contention on performance of applications.

4.2.1 CMT for Cache-Based Scheduling

In the past, researchers have attempted to use IPC to determine cache sensitivity of workloads. However, *Figure 15* shows that average workload IPC is a poor predictor of cache sensitivity. With CMT, cache sensitivity can now be more directly measured. In *Figure 16,* an example usage of CMT for cache based scheduling involves building occupancy-performance curves which provide a direct understanding of performance sensitivity of applications with respect to cache occupancy. *Figure 16* shows these curves for selected applications and demonstrates that cache occupancy is a good predictor of application performance sensitivity. It is possible to use the unique insight that CMT provides to more efficiently group applications for better cache utilization (e.g., a sensitivity crossover point where an additional 5% performance per MB of cache is considered to be the point of decreasing return on investment, which in turn provides an ideal operating point, and allows better scheduling assuming that we can schedule applications to provide enough cache to reach the maximum efficiency point). This insight can also help guide workload tuning to last-level cache size and select the optimal partition of data to allocate to each thread running on the processor socket.

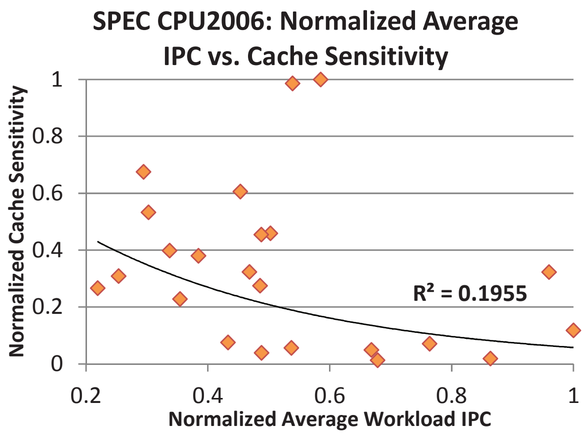


Figure 15. Avg. IPC vs Cache Sensitivity

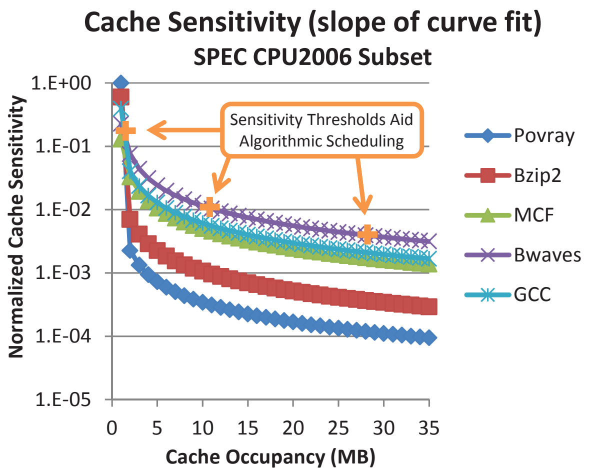


Figure 16. Cache Sensitivity Thresholds for scheduling

Rebalancing studies (e.g. *Figure 17*) show that an overall system throughput improvement is possible by employing CMT data and identifying applications that should be grouped together. The CMT-enabled platforms provide an opportunity for future research on policies used for such improvements and experimentation with online heuristics.

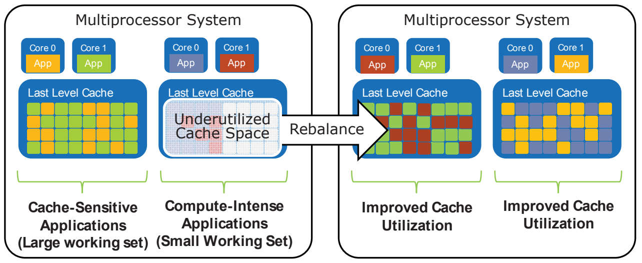
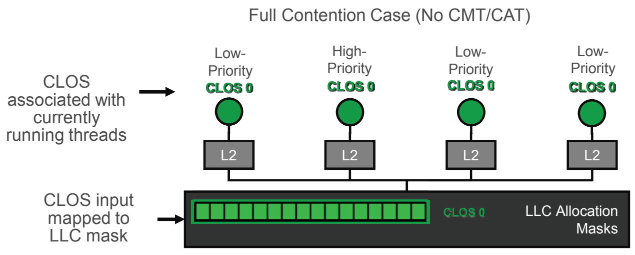
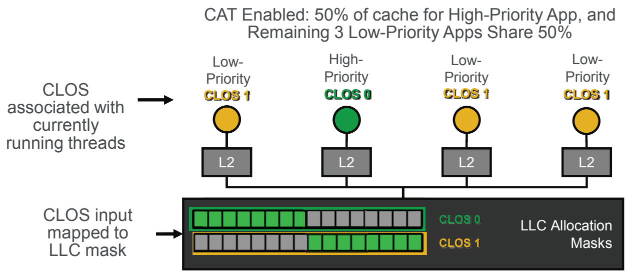

Multiprocessor System
Core 0
Core 1
App
Last Level Cache
Underutilized Cache Space
Rebalance
Cache-Sensitive Applications (Large working set)
Compute-Intense Applications (Small Working Set)
Improved Cache Utilization
Improved Cache Utilization

Figure 17. Rebalancing cache-intense apps across sockets for system throughput increase [3-4% w/ simple heuristics]


Full Contention Case (No CMT/CAT)
CLOS associated with currently running threads
Low-Priority
High-Priority
CLOS 0
L2
CLOS input mapped to LLC mask
CLOS 0
LLC Allocation Masks

(a) L3 Cache without Cache Allocation Technology


CAT Enabled: 50% of cache for High-Priority App, and Remaining 3 Low-Priority Apps Share 50%
CLOS associated with currently running threads
Low-Priority
High-Priority
CLOS 1
CLOS 0
L2
CLOS input mapped to LLC mask
CLOS 0
CLOS 1
LLC Allocation Masks

(b) L3 Cache with Cache Allocation technology

Figure 18. Illustration of CAT Usage (partitioning)

4.3 CAT Data and Use Cases (HPC, Cloud)

Once performance baseline data and cache sensitivity curves have been collected through application monitoring and CMT, cache space can be reallocated using Cache Allocation Technology as required. For instance, the cache can be partitioned to reduce contention. Figure 18(a) shows an example case without QoS (no CAT), where the system boots in CLOS[0], which has full access to the cache. Through the use of CAT (Figure 18(b)), the lower priority applications are assigned to CLOS[1], which has been configured with a smaller cache mask indicating 50% of the L3 cache is reserved for the high-priority *bzip2* application, and the remaining threads share the remaining 50% of the cache. Note that the diagram has been simplified to show only three lower-priority applications, not the thirteen lower-priority applications which are present on the thirteen other cores on the CMP processor.

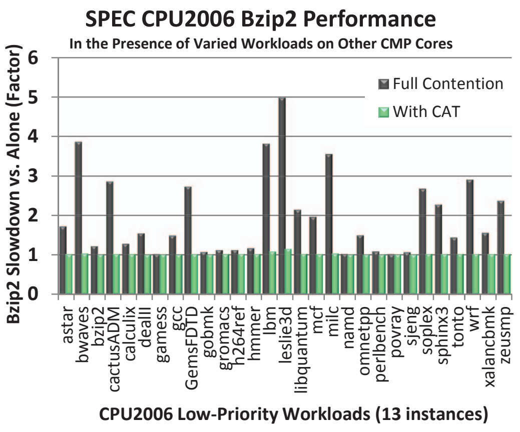
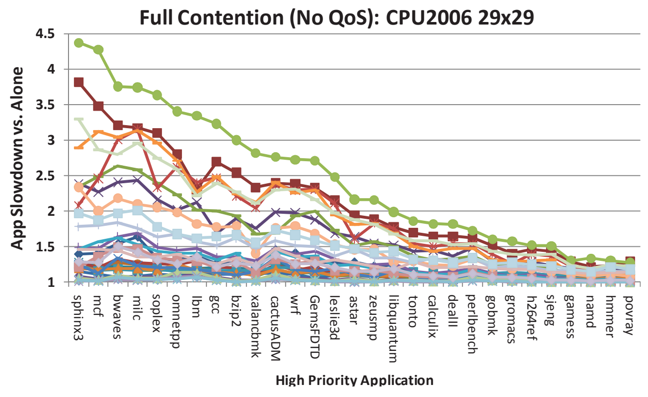


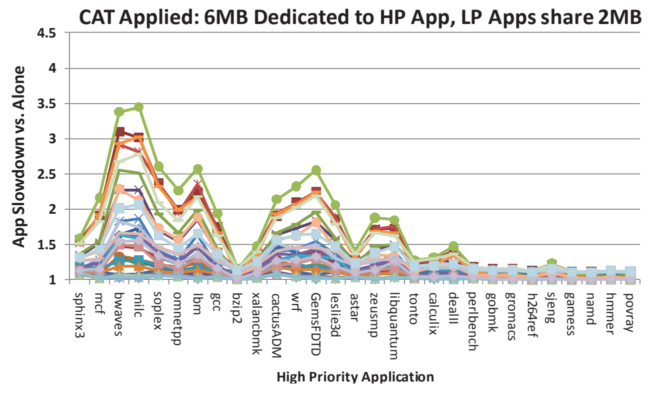
Figure 19. CAT restores the performance determinism of Bzip2 in addition to restoring performance

Figure 19 shows the data illustrating the benefits of Cache Allocation Technology. Using CAT in the L3 cache as described above, *bzip2* is able to easily fit its working set in the reserved 17.5MB (50%) of the cache. Since the "slowdown vs. alone" axis is already a ratiometric scale we can calculate the coefficient of variation before and after CAT to compare the results empirically. In the full contention case the coefficient of variation is 0.53, implying that 53% performance variation can be expected in a typical case. By contrast, when CAT is enabled the performance variation is reduced to the nearly flat set of green bars shown in *Figure 19*, and the coefficient of variation shrinks to 0.03. Thus, in the case of this cache-sensitive application, applying CAT to reduce thread contention resulted in an 18x reduction in performance variability for this workload, restoring predictable performance. In these studies CPU2006 provides a convenient proxy for HPC workloads as many of the floating-point intensive workloads are drawn from the scientific and HPC workload domains. Since the workload suite also provides a diverse set of applications, this helps emulate a cloud environment where many workloads may be consolidated from many different customers, each with different workload characteristics.

Experiments were also conducted with all workloads in the SPEC CPU2006 suite (beyond *bzip2*). *Figure 20(a)* illustrates the results for four applications run simultaneously and shows that the performance slowdown for a candidate application can be as high as 4.5X without Cache Allocation Technology. By using CAT, this performance slowdown can be addressed since we can partition the low priority applications in a smaller partition of the cache. *Figure 20(b)* shows the benefits of achieving this partitioning using CAT, resulting in much lower performance slowdown. While most cases show significant benefit, it should be noted that some cases still have high slowdown due to memory bandwidth contention.



(a) Full Contention without CAT



(b) with CAT

Figure 20. Addressing Contention with CAT

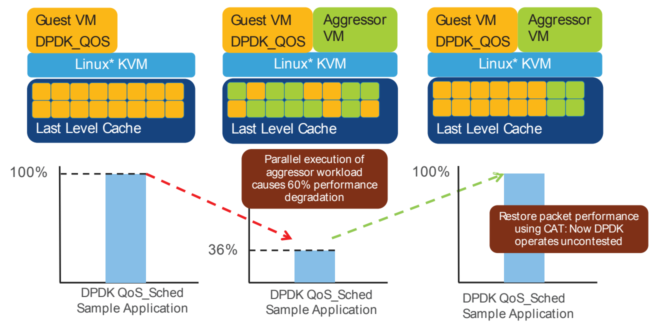

Guest VM DPDK_QOS
Aggressor VM
Linux* KVM
Last Level Cache
100%
36%
Parallel execution of aggressor workload causes 60% performance degradation
Restore packet performance using CAT: Now DPDK operates uncontested
DPDK QoS_Sched Sample Application

Figure 21. Restoring performance of a sample comms workload using CAT in presence of streaming applications

4.4 Communications-centric Workloads

Beyond SPEC results, CAT can provide benefits in a variety of communications and virtualized scenarios. Since communication between nodes is quite important in HPC a sample highly optimized communication workload is selected. This experiment utilizes the Intel Data Plane Development Kit (DPDK) [18] to demonstrate a virtualized Network Function. In this example the DPDK QoS sample application enqueues and dequeues packets between various stages before forwarding them to the network interface. In this case virtualization is used for application portability, but the same results are evident when running without virtualization. When executing the VM/application without any additional contending Virtual Machine or application running in the host there is very little cache pollution enabling the QoS application to scale to 11 Million Packets per Second (Mpps) with 64 byte packet size. *Figure 21* shows this base case in left-most configuration and chart.

When a second virtual machine (containing an aggressor workload – STREAM benchmark) is launched, the performance degradation is 64% due to cache contention (shown in middle chart and configuration in *Figure 21*). But this performance issue can be addressed using Cache Allocation Technology. In the third test scenario two classes of service have been created. The low priority class of service utilizes half of the L3 cache capacity mask. The high priority class of service also utilizes half of the capacity mask but is isolated. The aggressor virtual machine is associated with the low priority CLOS. The DPDK QoS sample application is assigned the high priority CLOS. Figure 21 (rightmost chart and config) shows the benefit of CAT in restoring the performance of the DPDK application.

5. Related Work

Over a decade now, studies have been available [2, 4, 7, 10, 13, 28] showing the impact of cache/memory contention on performance isolation, SLAs and overall throughput. Early in the decade, Iyer [10] initially introduced the QoS problem in CMP platforms and showed that priority information can be passed from systems software to hardware to control contention by resource partitioning and allocation. Around the same time, Kim et al [13] presented fairness issues in cache sharing in CMP platforms. Chandra et al [2] followed up with mechanisms for predicting this inter-thread contention. Subsequently, many researchers [1, 5, 6, 7, 11, 12, 15, 17, 20, 21, 22, 23, 24, 27] have proposed solutions for these issues and demonstrated using simulation and characterization measurements that these could be effective. For example, Rafique et al [21] proposed architectural support for shared cache management with quota-based approach. Qureshi et al proposed a utility-aware approach [20] to partitioning the cache space between applications. Jaleel et al [12] and improved on this by proposing adaptive insertion policies. Other researchers [6, 16, 22, 26] investigated the same issues for memory contention and proposed solutions as well. Researchers also embarked upon OS policies and solutions to address these issues and also studied system-level issues. Fedorova et al [3] studied performance isolation and employed operating system scheduling techniques to mitigate them. Knauerhase et al [14] studied use of OS observations to improve multi-core performance. Recently, Mars et al [15], Tang et al [26] and Zhang et al [29] studied the implications of cache/memory contention in datacenters including the issues around throughput and latency sensitivity.

All of the above works were important to explore the space of QoS and shared cache management overall and highlighted the key issues and considerations when developing an architecture. In this paper, we show how we productized key concepts from our research (cache monitoring and allocation) and developed technologies in real products and enabled them commercially to begin to address the shared cache QoS problem. These implementations in the Intel Xeon processor E5-2600 v3 family can be used by researchers and industry practitioners to further explore QoS. Our case studies in this paper highlight potential benefits and considerations when using these techniques.

6. Summary and Future Work

Quality of service has been a significant area of research for over a decade now. Our work on this subject started in 2004 when we introduced the QoS problem as well as potential solutions in early research papers. In this paper, we described two cache QoS technologies, *Cache Monitoring Technology* and *Cache Allocation Technology,* that have gone from research to reality in Intel Xeon processor E5-2600 v3 family which powers high performance computing clusters, cloud datacenter servers and in many other usages. We discussed the key considerations in enabling productization of these technologies. We also discussed the key system-level and software interfacing considerations for these technologies. Using the Intel Xeon processor E5-2600 v3 family we showed experimental data for the technologies highlight the benefits of cache QoS. We believe the platform now commercially available provides a rich QoS infrastructure for researchers and industry practitioners to utilize for many usage scenarios and enables new advanced resource management heuristic research and new workload management policies. In future work, there are several other important shared resources in the platform that require QoS support. We have shown how shared last-level cache contention can be addressed thus far and believe there is more research and development for other similar shared platform resources and policies to adaptively managing them across hardware and software.

Acknowledgements: We would like to thank numerous researchers, architects and designers at Intel who directly worked and/or provided input on exploring QoS techniques and its design/implementation.

Notices:

Software, benchmarks and workloads used in this paper are only for behavioral performance testing. The data and results in this paper are intended to provide understanding of some behavioral characteristics only and not absolute performance of platforms or features.

References

[1] K. Aisopos, J. Moses, R. Illikkal, R. Iyer, D. Newell, “PCASA: Probabilistic Control-Adjusted Selective Allocation for Shared Caches,” Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2012.

[2] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In HPCA ’05, Washington, DC, USA, 2005.

[3] A. Fedorova, M. Seltzer, and M. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. PACT ’07, Sep 2007.

[4] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam. Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. In SOCC ’11,Oct 2011.

[5] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for providing quality of service in chip multi-processors. MICRO 40, Dec 2007.

[6] A. Herdrich, R. Illikkal, R. Iyer, D. Newell, et al. Rate-based QoS techniques for cache/memory in CMP platforms. ICS ’09, Jun 2009.

[7] L. Hsu, S. Reinhardt, R. Iyer, S. Makineni, Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. 15th International Conference on Parallel Architectures and Compilation Techniques, PACT 2006.

[8] Intel Architecture and Processor Identification with CPUID Model and Family Numbers, https://software.intel.com/en-us/articles/intel-architecture-and-processor-identification-with-cpuid-model-and-family-numbers

[9] Intel SDM – Vol 3b., CH 17.14-17.15, Intel CMT, CAT technology.

[10] R. Iyer, “CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms”, Int’l Conference on Supercomputing (ICS), 2004.

[11] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS policies and architecture for cache/memory in cmp platforms. SIGMETRICS ’07, Jun 2007.

[12] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr, and J. Emer. Adaptive insertion policies for managing shared caches. PACT ’08, Oct 2008.

[13] S. Kim, D. Chandra and Y. Solihin, Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, PACT 2004.

[14] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS observations to improve performance in multicore systems. IEEE Micro, 28:54–66, May 2008.

[15] J. Mars, L. Tang, R. Hundt, K. Skadron and M. L. Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), MICRO-44, 2011

[16] K. Nesbit, N. Aggarwal, J. Laudon, and J. Smith. Fair queuing memory systems. MICRO-39, 2006.

[17] K. Nesbit, et al. Multicore resource management. Micro, IEEE, 28(3):6 – 16, 2008.

[18] Packet Processing -- Intel® DPDK, https://01.org/packet-processing

[19] Product Brief: Intel® Xeon® Processor E5-2600 v3 Family: http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-e5-brief.pdf

[20] M. Qureshi and Y. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. MICRO 39, Dec 2006.

[21] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for OS-driven cmp cache management. PACT ’06, Sep 2006.

[22] N. Rafique, W.-T. Lim, and M. Thottethodi. Effective management of dram bandwidth in multicore processors. PACT 2007, 2007.

[23] S. Srikantaiah, et al. A case for integrated processor-cache partitioning in chip multiprocessors. SC ’09, Nov 2009.

[24] S. Srikantaiah, et al. Adaptive set pinning: managing shared caches in chip multiprocessors. ASPLOS XIII, Mar 2008.

[25] Standard Performance Evaluation Corporation. SPEC Benchmark Suite. http://www.spec.org

[26] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *ISCA ’11*, New York, 2011.

[27] Y. Xie and G. Loh. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. *ISCA ’09*, Jun 2009.

[28] C. Xu, X. Chen, R. Dick, and Z. Mao. Cache contention and application performance prediction for multi-core systems. In *ISPASS 2010*, march 2010.

[29] Y. Zhang, M. Laurenzano, J. Mars, and L. Tang. SMiTe: Precise QoS Prediction on Real System SMT Processors to Improve Utilization in Warehouse Scale Computers. 47th IEEE/ACM International Symposium on Microarchitecture (MICRO), MICRO-47, 2014

[30] L. Zhao, R. Iyer, R. Illikkal, J. Moses, S. Makineni, D. Newell, "CacheScouts: Fine-Grain Monitoring of Shared Caches in CMP Platforms", *PACT*, 2007

[31] John D. McCalpin, “STREAM: Sustainable Memory Bandwidth in High Performance Computers”, http://www.cs.virginia.edu/stream/

[32] http://www.top500.org/lists/2014/11/

[33] Data Center Infrastructure Built on the Intel® Xeon® Processor E5 v3 Family, http://www.intel.com/content/www/us/en/cloud-computing/xeon-e5-products-real-world-guide.html