Improving QoS and Utilisation in modern multi-core servers with Dynamic Cache Partitioning

Ioannis Papadakis
School of ECE, NTUA
ypap@cslab.ece.ntua.gr

Konstantinos Nikas
School of ECE, NTUA
knikas@cslab.ece.ntua.gr

Vasileios Karakostas
School of ECE, NTUA
vkarakos@cslab.ece.ntua.gr

Georgios Goumas
School of ECE, NTUA
goumas@cslab.ece.ntua.gr

Nectarios Koziris
School of ECE, NTUA
nkoziris@cslab.ece.ntua.gr

ABSTRACT

Co-execution of multiple workloads in modern multi-core servers may create severe performance degradation and unpredictable execution behavior, impacting significantly their Quality of Service (QoS) levels. To safeguard the QoS levels of high priority workloads, current resource allocation policies are quite conservative, disallowing their co-execution with low priority ones, creating a wasteful tradeoff between QoS and aggregated system throughput. In this paper we utilise the cache monitoring and allocation facilities provided by modern processors and implement a dynamic cache partitioning scheme, where high-priority workloads are monitored and allocated the amount of shared cache that they actually need. This way, we are able to simultaneously maintain their QoS very close to the levels of full cache allocation and boost the system's throughput by allocating the surplus cache space to co-executing, low-priority applications.

Keywords

Co-scheduling; Cache partitioning; Cache management; QoS

1. INTRODUCTION

Multi-core systems have become the norm for high performance servers that are widely used in both HPC and cloud datacenters. These systems encapsulate several cores that share critical resources, such as cache space and memory bandwidth. When applications are executed simultaneously on such a system, contention for shared resources impacts the quality of service (QoS) and can lead to performance degradation.

To address this problem, researchers have mainly developed two orthogonal approaches. The first one extends the schedulers operating at different levels, from inside a single server [3, 5, 11, 12, 18–20, 22, 27, 30] up to a supercomputer [6], datacenter or cloud environment [7–9, 21, 29], to deal with contention. These contention-aware schedulers typically classify applications based on their utilisation of one or more of the shared resources. The classification is then used to identify co-schedules that mitigate contention and maximise the overall throughput or maintain performance fairness. The second approach investigates ways to effectively partition the shared resources among the concurrently running applications [10, 14, 16, 17, 23–26, 28]. Most of these works focus on the shared cache space and attempt to manage its allocation to the applications in an attempt to isolate or optimise their performance.

Nevertheless, until today, supercomputers and cloud datacenters do not employ any of the above solutions. Instead, in order to guarantee QoS, they typically do not allow co-execution of applications on a multi-core server, leaving a subset of cores idle to avoid any interference [6, 8, 29].

Intel® has recently developed and released as part of its latest Xeon® processors that power server platforms, the Intel® Resource Director Technology (RDT) [2], a hardware framework to monitor and manage shared resources. The goal of our work is to leverage this technology to provide a mechanism that alleviates the effects of contention, making the co-execution of applications a viable choice even when QoS needs to be guaranteed.

To this end, we implement a dynamic cache partitioning scheme that monitors the throughput of a high-priority application and allocates to it the amount of shared cache that it actually needs, safeguarding its QoS as the performance penalty compared to full cache allocation is minimised to 5% on average. At the same time, the cache surplus is allocated to the co-executing low-priority applications, boosting their performance by a factor of 5× compared to when the cache is fully assigned to the high-priority workload, thus increasing the aggregated system throughput.

2. BACKGROUND & MOTIVATION

2.1 Architectural Support

Intel® RDT provides the necessary hardware support to monitor and manage the last level cache (LLC) and the memory bandwidth [15]. This is achieved through four technologies:

- Cache Monitoring Technology (CMT), which monitors the usage of the shared LLC.
- Cache Allocation Technology (CAT), which enables the management of the distribution of the shared LLC

COSH-VisorHPC 2017 Jan 24, 2017, Stockholm, Sweden

ISBN 978-3-00-055564-0
DOI: 10.14459/2017md1344298

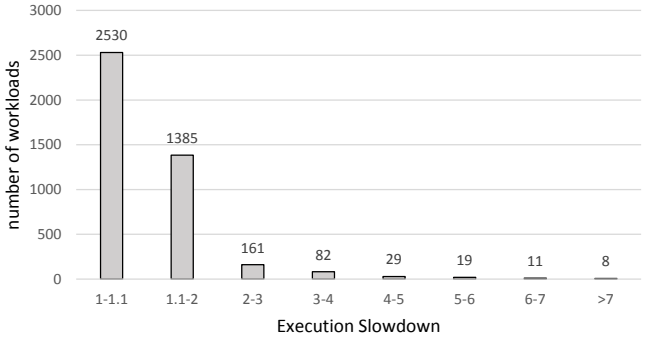


Figure 1: Distribution of HP application's slowdown when running together with 21 LP applications.

among the concurrently running applications. The allocated cache areas can overlap or be isolated.

- Memory Bandwidth Monitoring (MBM), which monitors the usage of the memory links.
- Code and Data Prioritization (CDP), which enables separate control over code and data placement in the LLC.

All these technologies can be programmed and utilised via Model Specific Registers (MSR) on a hardware thread basis. Therefore, they can be used in all modern Operating Systems, which provide APIs to read and write the MSRs.

2.2 Motivation

To guarantee QoS, supercomputers and datacenters typically avoid placing multiple applications on the same multi-core server [6, 8, 29]. To showcase the problems that arise when attempting to utilise all the processing cores of a server, we execute multiprogrammed workloads on an Intel Xeon E5-2699 v4 server. The system is configured with a 2.2 GHz processor with 22 cores, SMT disabled, 55MB L3 cache (LLC), and 64GB of memory.

We employ 11 benchmarks from the Parsec 3.0 benchmark suite [4] (serial versions) and 28 benchmarks from the SPEC CPU 2006 suite [13], 9 of which can be used with multiple inputs, bringing the total number of applications to 65. Each multiprogrammed workload is created by nominating one of the 65 benchmarks as a "High-Priority" application (HP) and placing it on one of the cores of the server. Then another benchmark is selected as the "Low-Priority" application (LP) and 21 instances of it are placed on the remaining cores of the system. Every time an LP instance finishes, we restart it to make sure that the system is always full until HP finishes its execution.

We execute all the 4,225 workloads and compare the execution time of the HP application to when it is executed alone on the system. As shown in Figure 1, in around 60% of the workloads, HP suffers at most 10% slowdown, while in the remaining 40% there is a significant impact on the QoS. In around 33% of the cases, HP's execution time can double compared to when running alone, while in 310 workloads co-execution thrashes HP, causing its execution time to increase even more than 8 times in some cases.

It is evident that in order to increase the utilisation of the platforms while guaranteeing QoS, we must diminish the impact of co-execution due to resource contention.

3. CACHE-BASED QoS

The CAT technology has been shown to be able to restore the performance of an HP application in multiprogrammed workloads by reserving for that application a percentage of the shared LLC while restraining all other applications in the remaining cache space [15]. However, to leverage this technology on a multi-core platform of a supercomputer or a datacenter, one would need to know the exact amount of cache space that the HP application requires to be isolated from the others.

This amount depends on the behaviour of the actual HP application as well as on the behaviour of its co-runners. Therefore, the system would need to know at any given moment which applications are running, how they behave and how much space HP requires in the currently executed workload.

Alternatively, a system could conservatively limit all the LP applications in the minimum possible cache space, i.e., a single way of the set-associative LLC, reserving the rest of the cache space for the HP application. We refer to this conservative, static allocation as "Cache-Takeover QoS" (CT-QoS). CT-QoS is expected to provide the best possible performance for HP, at the expense however of the LP applications. At the same time, there is a high chance that HP does not utilise all the allocated cache space and it could have achieved similar performance with less cache. Furthermore, the cache requirements of an application typically vary during its execution, making the static assignment of cache space for the whole execution of the application a poor choice.

In this paper we propose a mechanism that dynamically adapts the HP application's LLC allocation, trying to match its cache requirements at any moment of its execution. We refer to this approach as "Dynamic Cache-Partitioning QoS" (DCP-QoS). DCP-QoS attempts to guarantee similar performance for the HP application to that achieved by CT-QoS, while at the same time allowing more cache space to be used by the LP applications. This extra space allocated to LPs, will enable them to achieve higher performance compared to CT-QoS, thus increasing the aggregated throughput of the system.

3.1 DCP-QoS

The flow chart of our scheme is presented in Figure 2. DCP-QoS makes initially the same conservative choice as CT-QoS, i.e., all the LP applications are restrained within the minimum possible cache space. Assuming a system with an N-way associative LLC, DCP-QoS allocates $N-1$ ways to HP and only 1 way to LPs. It then waits for HP to reach a steady state before attempting any changes.

Once our scheme detects that the performance of HP is stable, it starts gradually reducing its allocation by one way, which gets assigned to the LP applications. When the cache space has been reduced to a point where HP's performance is no longer stable, our mechanism increases HP's allocation by a single way. If this does not make the performance stable again, our mechanism resets, i.e., allows HP to takeover the LLC and limits LPs to a single way. If it does, then we assume that we have reached a "Balanced state", where the HP application has enough cache space allocated to enable it to achieve similar performance to when running with $N-1$

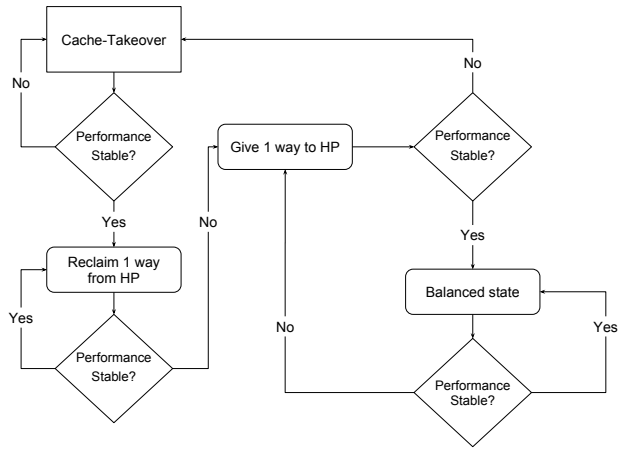


Figure 2: Flowchart of the proposed DCP-QoS mechanism.

ways.

While in the Balanced state, our mechanism continues to monitor HP's performance. If it detects a change, then it increases HP's allocation by a single way and checks whether the extra space was enough to make the performance stable again. If not, our mechanism resets and HP receives again $N-1$ ways.

3.1.1 Determining performance stability

To determine whether the performance of HP is stable or not, we define a monitoring period with length T and measure the IPC HP obtained during that time. As IPC is expected to fluctuate between two monitoring periods, we define that IPC_{t_1} and IPC_{t_2} are considered equal if :

$$(1-a) * IPC_{t_1} \leq IPC_{t_2} \leq (1+a) * IPC_{t_1} \qquad (1)$$

At the end of each time interval i, we compare HP's IPC, IPC_i, with its IPC during the previous two periods, IPC_{i-1} and IPC_{i-2}. If:

- $IPC_{i-1} = IPC_{i-2}$ and $IPC_i = IPC_{i-1}$, then the performance is considered stable.
- $IPC_{i-1} = IPC_{i-2}$ and $IPC_i \neq IPC_{i-1}$, then the performance is considered not stable.
- $IPC_{i-1} \neq IPC_{i-2}$ and $IPC_i \neq IPC_{i-2}$, then the performance was found to be not stable at the end of the previous monitoring interval. This caused our mechanism to allocate one more cache way to the HP application, without however succeeding in restoring the performance. Therefore, the performance is determined to be unstable again, leading our mechanism to reset.
- $IPC_{i-1} \neq IPC_{i-2}$ and $IPC_i = IPC_{i-2}$, then the performance that was found to be unstable before has now been restored again to its previous levels. Therefore, we consider the performance to be stable and our mechanism has reached the Balanced state.

As is evident by Equation 1, our mechanism can deduce that performance is not stable and reset, even when HP's IPC has increased. This design choice is justified by the fact that a significant change in the IPC, regardless of whether it is increased or not, could imply a phase change of the application. Therefore, our mechanism resets and starts looking for the new Balanced state from the beginning. Note that different metrics could also be used for driving the mechanism, such as LLC misses; we leave the exploration of such options for future work.

3.1.2 Cache Utilization Monitoring

We further augment DCP-QoS to take into consideration also the cache metrics offered by the Intel technologies. During each monitoring period T, using CMT and MBM, we acquire n samples of how much space HP occupies in the LLC and how much memory bandwidth it uses, recording only the maximum cache occupancy and the maximum bandwidth utilisation.

At the end of T, once the stability of the performance has been determined and a decision regarding HP's allocation has been made, our mechanism compares the recorded utilisation of the memory channel by HP to a predefined threshold. If the utilisation is higher than the threshold, we can speculate that HP tends to fully utilise its allocated space and proceed with modifying its allocation based on the performance stability, as depicted in Figure 2.

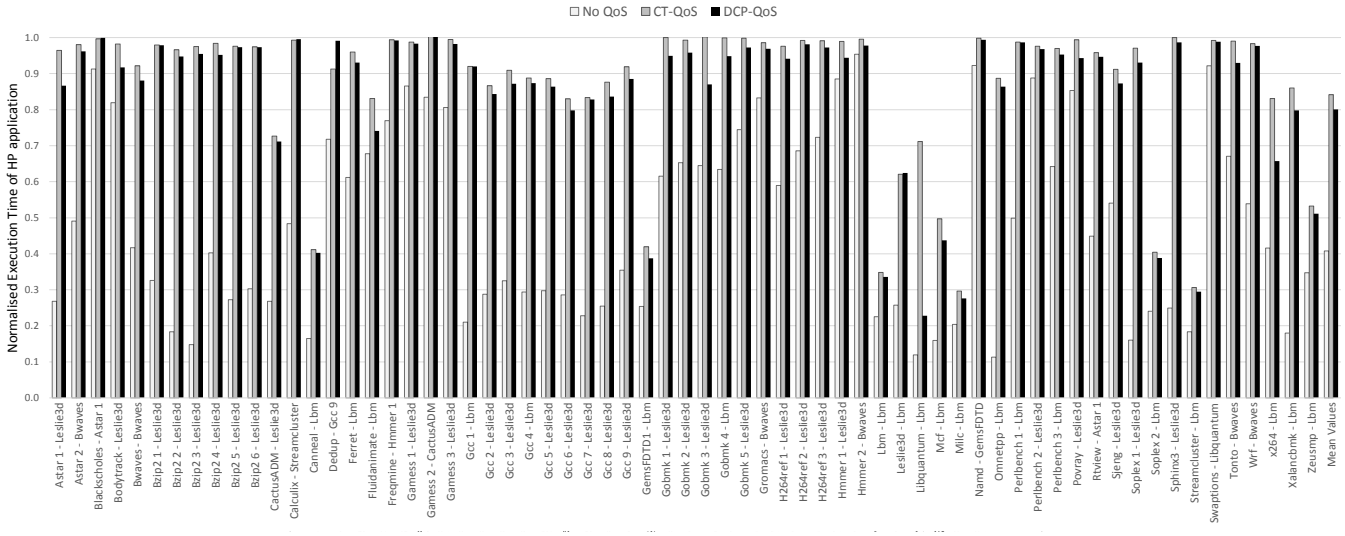
If, however, the utilisation is below the predefined threshold, we can deduce that HP is content with its LLC allocation, as it does not suffer many LLC misses. At this point, we compare the recorded maximum LLC occupancy to the cache space allocated to HP by our mechanism. If HP is found to occupy less space than allocated, then the redundant cache ways are removed from HP and assigned to LPs. That way, we are able to reduce HP's allocation more aggressively than by removing just one cache way at the end of every monitoring period.

Finally, we use the memory bandwidth utilisation to avoid unnecessary resets while in the Balanced state. Specifically, if our system has reached the Balanced state and the performance is found unstable, before resetting, we look at the recorded memory bandwidth measurement of the last monitoring period. If it is found to be less than the predefined threshold, then we speculate that HP will not utilise the extra space provided by a reset and choose to remain in the Balanced state.

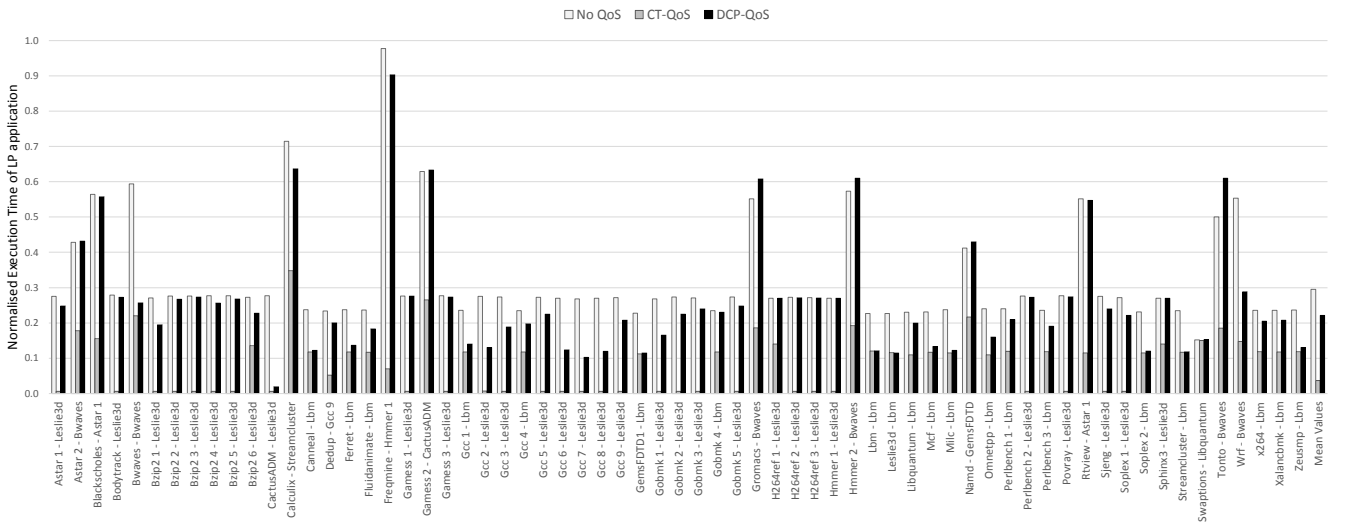
4. EVALUATION

Processor	Intel Xeon E5-2699 v4 (Broadwell)
	22 cores, 2.2GHz, SMT disabled
Memory	64GB DDR4
Memory Bandwidth	153.6 Gbps per channel
LLC	55MB, 20-way set associative
DCP-QoS	$a = 0.05$
	$T = 100ms$
	$n = 10$
	$bandwidth\_threshold = 27.5Mbps$

Table 1: System Configuration



(a) HP



(b) LP

Figure 3: Execution time of the (a) HP application and (b) LP applications, normalized to when each application runs alone on the system (higher is better). DCP-QoS maintains the QoS of the HP application very close to the levels of CT-QoS and at the same time improves the performance for the LP applications by allocating the surplus cache space to them.

4.1 Methodology

We implement DCP-QoS by extending the Intel RDT Software Package v0.1.5 [1], an open source stand-alone library that provides support for CMT, MBM, CAT, and CDP. As the library controls these technologies via the MSRs, it runs in privileged mode. In addition, the library calculates various metrics such as IPC. The evaluation is performed on an Intel Xeon E5-2699 v4 processor. The details of the processor, together with the predefined parameters of DCP-QoS are presented in Table 1.

For the evaluation, we employ 11 benchmarks from the Parsec 3.0 benchmark suite [4] (serial versions) and 28 benchmarks from the SPEC CPU 2006 suite [13], 9 of which can be used with multiple inputs, bringing the total number of applications to 65. We create multiprogrammed workloads by selecting one benchmark as the HP application and 21 instances of another as the LP applications. We first execute all the 4,225 possible co-executions on our system without any QoS mechanisms, allowing full contention for the shared resources. Based on the results, we select for each of the 65 benchmarks its worst co-execution, i.e., the LP co-runners that caused its performance to degrade the most.

We then use the selected 65 co-executions to compare the proposed DCP-QoS mechanism with two other configurations. First the "No QoS" configuration, where no QoS mechanism is used, hence the HP and the LP applications experience full contention on the LLC and memory channel. Second, the CT-QoS configuration, where 19 ways are statically allocated to the HP application and only 1 way is

allocated to the LP applications.

4.2 Results

Figures 3a and 3b show the execution time of the HP application and the LP applications respectively for the selected co-execution scenarios, normalized to when each application runs alone on the system; the last bars show the geometric mean.

The results with the No QoS configuration show that the lack of a QoS mechanism can severely harm the performance of the HP application due to the contention in the cache and the memory bandwidth with the LP applications, as was also shown in Section 2.2. This performance degradation can reach up to 90% and is on average 59%, compared to when running alone.

CT-QoS preserves QoS for the HP application by allowing them to execute on average by 84% compared to when running alone. However, CT-QoS achieves that QoS for the HP application at the cost of significantly sacrificing the performance of the LP applications and underutilising the shared resources. The reason is that CT-QoS pessimistically allocates 19 ways of the LLC for the HP application in a static fashion, even when less ways and hence cache space would have provided similar performance to the HP application. Indeed, CT-QoS degrades the performance of the LP application by 96% compared to when running alone.

Our proposed mechanism DCP-QoS enjoys almost the same QoS benefits that CT-QoS provides for the HP application, while increasing the performance of the LP applications too. More specifically, DCP-QoS achieves high performance for the HP application, by 80% on average compared to when running alone, and ensures QoS close to that of CT-QoS by less than 5%. In addition, DCP-QoS enables higher cache utilisation, increases the overall throughput, and improves the performance for the LP applications by 5× compared to the CT-QoS approach (from 4% to 22% compared to when running alone). Overall, DCP-QoS provides a good trade-off between QoS and utilisation, and enables the co-location of high and low priority workloads on the same server.

5. CONCLUSIONS AND FUTURE WORK

In this paper we experimented with the cache monitoring and partitioning facilities provided by modern multicore processors to test their efficacy in a co-scheduling scenario where a high priority process co-exists with a number of low priority ones. We devised a dynamic cache allocation scheme that monitors the execution behavior of the high priority process and adapts the cache size according to its demands. In this way, we were able to validate that it is possible to maintain the QoS levels of high priority processes to those of full cache allocation, while at the same time significantly boosting the system's throughput by providing more resources to the low priority processes.

We intend to extend our approach to more complex execution scenarios involving different mixtures of processes and priority levels, and combine the allocation decisions with workload characterization schemes to further improve its QoS and throughput.

6. ACKNOWLEDGMENTS

This research has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement no 732366 (ACTiCLOUD).

7. REFERENCES

[1] Intel RDT Software Package. https://github.com/01org/intel-cmt-cat.

[2] Intel Resource Director Technology. https://www-ssl.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html.

[3] M. Bhadauria and S. A. McKee. An approach to resource-aware co-scheduling for CMPs. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 189–199, New York, NY, USA, 2010. ACM.

[4] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[5] S. Blagodurov, S. Zhuravlev, and A. Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, 28(4):8:1–8:45, Dec. 2010.

[6] A. D. Breslow, L. Porter, A. Tiwari, M. Laurenzano, L. Carrington, D. M. Tullsen, and A. E. Snavely. The case for colocation of high performance computing workloads. *Concurrency and Computation: Practice and Experience*, pages 232–251, 2013.

[7] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 77–88, New York, NY, USA, 2013. ACM.

[8] C. Delimitrou and C. Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pages 127–144, 2014.

[9] C. Delimitrou and C. Kozyrakis. Hcloud: Resource-efficient provisioning in shared cloud systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, pages 473–488, 2016.

[10] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 335–346, New York, NY, USA, 2010. ACM.

[11] A. Haritatos, G. Gourmas, K. Nikas, and N. Koziris. A resource-centric application classification approach. In *Proceedings of the 1st COSH Workshop on Co-Scheduling of HPC Applications, COSH@HiPEAC 2016, Prague, Czech Republic, January 19, 2016.*, pages 7–12, 2016.

[12] A.-H. Haritatos, G. Goumas, N. Anastopoulos, K. Nikas, K. Kourtis, and N. Koziris. LCA: A memory link and cache-aware co-scheduling approach for CMPs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 469–470, New York, NY, USA, 2014. ACM.

[13] J. L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[14] A. Herdrich, R. Illikkal, R. Iyer, D. Newell, V. Chadha, and J. Moses. Rate-based qos techniques for cache/memory in cmp platforms. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 479–488, New York, NY, USA, 2009. ACM.

[15] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer. Cache QoS: From concept to reality in the Intel® Xeon® processor e5-2600 v3 product family. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 657–668, 2016.

[16] R. R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. R. Hsu, and S. K. Reinhardt. Qos policies and architecture for cache/memory in CMP platforms. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2007, San Diego, California, USA, June 12-16, 2007*, pages 25–36, 2007.

[17] A. Jaleel, W. Hasenplaugh, M. K. Qureshi, J. Sebot, S. C. S. Jr., and J. S. Emer. Adaptive insertion policies for managing shared caches. In *17th International Conference on Parallel Architecture and Compilation Techniques (PACT 2008), Toronto, Ontario, Canada, October 25-29, 2008*, pages 208–219, 2008.

[18] A. Jaleel, H. H. Najaf-abadi, S. Subramaniam, S. C. Steely, and J. Emer. Cruise: Cache replacement and utility-aware scheduling. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 249–260, New York, NY, USA, 2012. ACM.

[19] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 220–229, New York, NY, USA, 2008. ACM.

[20] K. Kumar Pusukuri, R. Gupta, and L. N. Bhuyan. ADAPT: A framework for coscheduling multithreaded programs. *ACM Trans. Archit. Code Optim.*, 9(4):45:1–45:24, Jan. 2013.

[21] 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. In *MICRO*, pages 248–259, 2011.

[22] A. Merkel, J. Stoess, and F. Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 153–166, New York, NY, USA, 2010. ACM.

[23] K. J. Nesbit, M. Moretó, F. J. Cazorla, A. Ramírez, M. Valero, and J. E. Smith. Multicore resource management. *IEEE Micro*, 28(3):6–16, 2008.

[24] K. Nikas, M. Horsnell, and J. D. Garside. An adaptive bloom filter cache partitioning scheme for multicore architectures. In *ICSAMOS*, pages 25–32, 2008.

[25] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, pages 423–432, 2006.

[26] S. Srikantaiah, M. T. Kandemir, and M. J. Irwin. Adaptive set pinning: managing shared caches in chip multiprocessors. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, pages 135–144, 2008.

[27] L. Tang, J. Mars, and M. L. Soffa. Contentiousness vs. sensitivity: improving contention aware runtime systems on multicore architectures. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 12–21, New York, NY, USA, 2011. ACM.

[28] Y. Xie and G. H. Loh. Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 174–183, New York, NY, USA, 2009. ACM.

[29] H. Yang, A. D. Breslow, J. Mars, and L. Tang. Bubble-flux: precise online qos management for increased utilization in warehouse scale computers. In *The 40th Annual International Symposium on Computer Architecture, ISCA'13, Tel-Aviv, Israel, June 23-27, 2013*, pages 607–618, 2013.

[30] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. *SIGPLAN Not.*, 45(3):129–142, Mar. 2010.