LCN : A prediction-based contention-aware scheduler for multithreaded applications

Raptis Dimos-Dimitrios *

National Technical University Of Athens

Abstract

In this paper, we present a contention-aware scheduler for multi-threaded applications that is based on predictions regarding the scalability of applications. Our approach is comprised of 3 components : a classification scheme, a prediction model and a scheduling algorithm. The classification scheme is used to detect which resources of the system are primarily affected by contention, thus limiting the scalability of the application. A different prediction model for each class is used to predict the optimum allocation of resources, which results in the best tradeoff between scaling and resource management. The scheduling algorithm uses the prediction results to schedule the applications in a way that the performance degradation due to contention is minimized. The proposed scheduler is compared with other state-of-the-art schedulers in terms of throughput and fairness, since it is simple and easily integratable in modern operating systems.

I. Introduction

The technological progress in the field of computer hardware has been increasing rapidly during the last decades. However, the exponential difference between the progress of the computing power and the memory speed imposes a bottleneck in the ways we can leverage the bigger amount of available processing power [5]. The dominance of chip multi-processors (CMPs) has been followed by initiatives to take advantage of the various forms of parallelism, such as instruction-level parallelism or thread-level parallelism. This has been achieved through various frameworks, like openMP, Cilk, openMPI etc. Nonetheless, the shared memory hierarchy enforces various limitations to those attempts. Some problems arise due to memory bandwidth limitations and are handled by methods, such as improving data locality or improved prefetching [6], but as long as the working-set size of a job exceeds the size of the on-chip cache, the effectiveness of such approaches is quite limited. Another cause of performance degradation is the cache contention between different applications. Applications allocated to different cores of the same chip interfere with each other and the result of this interference on the overall performance depends on the data sharing patterns

*The author thanks the Computing Systems Laboratory of National Technical University of Athens for providing the necessary infrastructure for completing the required experiments and specifically A.Haritatos and N.Koziris for continuous supporting, consulting and guiding this research.

between the applications. Hardware or software-based methods have tried to analyse and improve data-sharing patterns, such as Utility Cache Partitioning [4] or Page Colouring [3]. However, the feasibility of those methods on modern operating systems are somewhat limited due to their requirement of additional hardware support or non-trivial changes to the virtual memory management system.

So, attempts are focused on implementing contention-aware schedulers that can detect and mitigate resource contention, since the current schedulers of modern operating systems are contention-unaware. Similar approaches handle applications either as single-threaded applications or as multi-threaded applications, where the allocation of resources is already predefined and the scheduler does not contribute to this step. Our approach handles applications as black boxes without having any knowledge regarding required resources, since the definition of the optimum amount of resources that should be allocated for ideal scaling is predicted by a component of the scheduler. Most contention-aware schedulers consist of 2 parts : a classification scheme defining performance degradation for combinations of co-scheduled application and a scheduling policy using those estimations to schedule the corresponding workload. Our approach follows this philosophy, but it also contains a prediction model as intermediate step, where the optimum number of resources that should be allocated to each application to avoid resource contention is estimated.

The classification scheme categorizes applications in categories based on whether they are memory-intensive or CPU-instensive. If the applications are memory-intensive, they are further categorized according to the memory hierarchy domain where contention is mainly observed. Afterwards, according to the class of each application, the suitable relationships of the prediction model are used to estimate an optimum number of cores that should be allocated in order to achieve the best tradeoff between scaling and resource management. The classification scheme has been implemented based on observations of previous research [1], showing that this classification is theoretically grounded and presents better results than other classification schemes, like animal classes [16] or color classes [17].

This scheduler has been tested with a workload of 17 applications spread amongst all categories. The scheduler is implemented in a user-space framework and evaluated on a Sandy Bridge architecture CMP. The results of the scheduler are compared with those of other state-of-the-art contention-aware schedulers, like LLC-MRB [13, 15] and LBB schedulers [13, 14] proposed by previous research, and with the current scheduler of Linux (CFS), which is contention-unaware. The schedulers are compared using total throughput and fairness as main criteria, with our approach presenting the best overall throughput. Furthermore, the prediction model has also been verified in a Nehalem architecture CMP for further validation. The architectures of the 2 systems used for experimentation can be seen in the following tables.

Table 1: *Sandy Bridge Architecture*

Cores	8
L1	Data Cache: private, 32KB Instruction Cache: private, 32KB
L2	private, 256KB
L3	shared, 32MB
Memory	64GB, DDR3, 1333MHz
OS	Debian Linux 6.0.6 with kernel 3.7.10

Table 2: *Nehalem Architecture*

Cores	4
L1	Data Cache: private, 32KB Instruction Cache: private, 32KB
L2	private, 256KB
L3	shared, 16MB
Memory	32GB
OS	Ubuntu 12.04.2 LTS

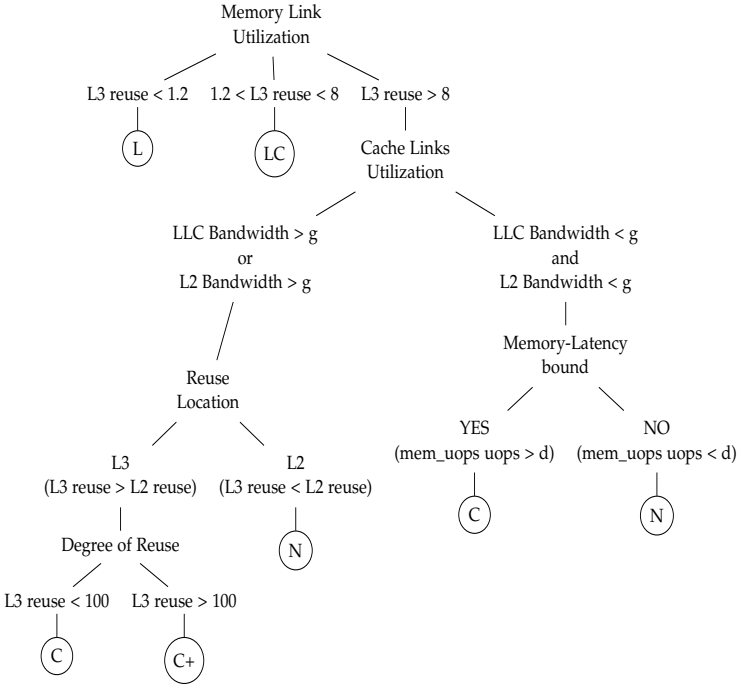
The rest of the paper is organized as follows : in Section II we define our classification scheme and analyse the interference between applications of different classes. Section III presents our prediction model and statistics regarding its correctness. Our scheduling algorithm and the statistical comparison between other schedulers are described in Section IV. Finally, Section V concludes the paper and proposes ideas for future research.

II. Classification Scheme

The purpose of our classification scheme is primarily to locate contention on private caches, shared last-level cache and memory link and secondarily to depend only on information that can be collected during the application's execution from the existing monitoring facilities of modern processors without requiring additional hardware enhancements. The applications are categorised in the following 4 main classes:

- Class L: applications with high usage of memory link, consuming a high percentage of memory bandwidth
- Class LC: applications that combine significant activity in last-level cache with considerable usage of memory link
- Class C: applications with really high usage of the last-level cache. Specifically, class C+ is a subcategory, where applications distribute their data usage between the shared and the private part of the memory hierarchy.
- Class N: applications whose activity is focused either in the private caches or within the core.

The applications are categorised based on specific hardware performance counters monitored during their execution. More specifically, the used counters are UNHLT_CORE_CYCLES, INSTR_RETIRED, LLC_MISSES, L1_LINES.IN, L2_LINES.IN, MEM_UOP_RETIRED.ALL and UOPS_RETIRED.ALL. Furthermore, we use OFFCORE_REQUESTS (0xB7, 0x01; 0xBB, 0x01) together with Intel's Performance Counter Monitor [9] utility to acquire information regarding bandwidth usage. The decision tree that is used for the classification scheme can be seen in the following diagram.



where the following relations have been used:

$$Memory\ bandwidth = (per\_core\_bw * 64)/(10^6)(GB/sec)$$
$$LLC\ bandwidth = (l2\_lines\_in * 64)/(10^6)(GB/sec)$$
$$L2\ bandwidth = (l1\_lines\_in * 64)/(10^6)(GB/sec)$$
$$L3\ reuse = LLC\ bandwidth/Memory\ bandwidth$$
$$L2\ reuse = L2\ bandwidth/LLC\ bandwidth$$
$$g = 0.35 * MaximumMemoryBandwidth$$
$$d = 0.25$$

III. Prediction Model

The next step after classifying an application is to use our prediction model in order to make an estimation of the maximum resources that can be allocated in an application without provoking resource contention and performance degradation. The design of the prediction model was based on statistical investigation described below combined with theoretic validation of the results. For each class of applications, a specific ratio of counters and a threshold value are defined. We accept as optimum the number of cores that when allocated to the application, this ratio gets the maximum possible value without surpassing the threshold value.

For applications belonging to class L, contention is mainly observed in the memory link, so the memory bandwidth is the crucial factor. The maximum memory bandwidth of the system has to be known and it can be measured using stream benchmark [7], as we did in our approach. If the memory bandwidth of the application when executed with 1 core has value Mem_1, then the used ration is $R = (Mem_1 * p)/(Maximum\ Memory\ Bandwidth)$ and the threshold value is $T = 1.15$. This is based on the fact that when allocating more cores to an L-class application, the memory bandwith consumed can be approximated from simple multiplication with the number of cores. However, this fact is quite valid, since class-L applications have low L3 reuse, so different cores of the application fetch data continuously from the memory. The threshold value is chosen after investigation, since performance degradation is observed after the memory link is fully occupied.

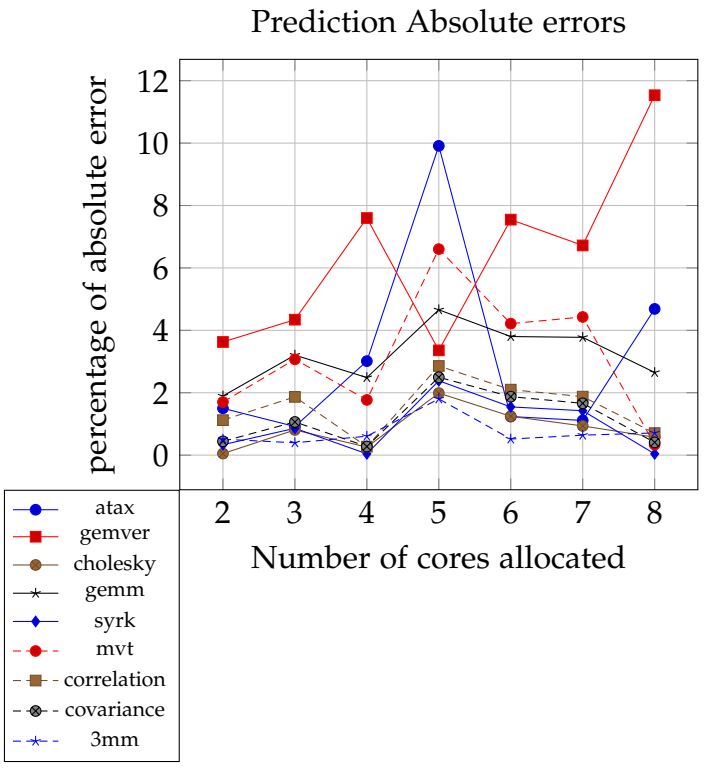
After experimentation with the linear regression model, we discovered that the scaling of applications belonging to classes C, C+, LC are linearly correlated with specific counters relative to the corresponding part of the memory hierarchy that suffers contention in each class. So, we examined all the related counters and for each class we found a combination that had a high correlation coefficient, reaching values around 0.95, which led us to create the following prediction relations. For LC applications, $f_{LC} = L2\ RFO\ Requests/L3\ reuse$ is correlated to the scaling of the application, while for C and C+ applications $f_C = L2\ shared\ allocated\ lines/Instructions\ Retired$ is the corresponding correlated combination. A theoretic explanation lies behind each combination for each class. LC applications that scale well should have a combination of low L2 RFO (Read-For-Ownership) requests and high L3 reuse, so that data are not invalidated between different cores and those few that are invalidated are fetched from LLC cache and not from memory, where a bigger penalty would occur. C applications that scale well should have low L2 shared lines proportionally to the instructions retired, so that there is low cache thrashing when allocating more cores. After examining a bunch of random applications, we ended up with 7 linear relationships for each class, one for each dfferent number of cores. By executing an application with 1 core and deriving the needed f_i, we can predict the scaling for p cores by using the p^{th} relationship. The used ratio is $R = (Scaling_{ideal}/Scaling) * 100$ and the threshold value is $T = 70$. It can be

argued that the threshold value is chosen somewhat arbitrarily, but it is theoretically ground meaning that the scaling must be between 30 percent divergence from the ideal. The ideal scaling is calculated as $Scaling_{ideal} = 1/p$. For instance, with 3 cores allocated the ideal scaling would be $1/3 = 0.333$. The threshold value can be set higher if we want to be stricter regarding the optimal scaling and contrariwise.Below, we can see the derived linear relations for LC,C applications (C+ are omitted for brevity).

$$
\begin{aligned}
Scaling &= 0.01799 * f_{LC} + 0.50119(2cores) \\
&= 0.025163 * f_{LC} + 0.34286(3cores) \\
&= 0.02846 * f_{LC} + 0.26028(4cores) \\
&= 0.03199 * f_{LC} + 0.21584(5cores) \\
&= 0.03404 * f_{LC} + 0.18296(6cores) \\
&= 0.036213 * f_{LC} + 0.1641(7cores) \\
&= 0.03751 * f_{LC} + 0.139699(8cores)
\end{aligned}
$$

$$
\begin{aligned}
Scaling &= 0.3447 * f_{C} + 0.4947(2cores) \\
&= 0.46974 * f_{C} + 0.34415(3cores) \\
&= 0.5155 * f_{C} + 0.2478(4cores) \\
&= 0.63609 * f_{C} + 0.22492(5cores) \\
&= 0.61403 * fC + 0.18127(6cores) \\
&= 0.65915 * f_{C} + 0.15864(7cores) \\
&= 0.6095 * f_{C} + 0.1263(8cores)
\end{aligned}
$$

The relations have also been tested in a Nehalem architecture successfully with the following observation. The coefficients of the linear regression model are all multiplied with $1/2$ and this can probably be interpreted by the fact that the machine with Nehalem architecture had shared last-level cache with half size compared to the caches of the Sandy Bridge machine. In the following, graphs we can see the errors in prediction for various benchmarks belonging to those 3 classes (LC,C,C+). We can notice that absolute errors do not exceed 15%, which is a quite optimistic number



As expected, N-class applications do not present significant resource contention and scale up to maximum resources allocated, since these applications restrict their activity either to the private part of the memory hierarchy or within the core, so the additional allocated cores do not interfere creating performance degradation. So, for N-class applications the maximum number of cores has been defined as optimum (8 in Sandy Bridge, 4 in Nehalem) and there was no deviation from this conclusion in the measurements.

IV. Scheduling Policy

The final step of the suggested scheduler is the algorithm used to co-schedule the applications. After having completed the first 2 steps for all the applications that need to be scheduled, each application is classified and the number of cores that should be allocated to it is defined. So, ap-

plications are separated in 4 lists, one for each class. Then, the following algorithm is used to co-schedule the applications. We have made the realistic assumption that no more than 2 application can be co-scheduled, so the algorithm attempts to co-schedule the applications solely in pairs. However, it can easily be extended to co-schedule using bigger combinations.

Co-Scheduling algorithm

```
Input: L,LC,C,N lists of applications

    for(i=0;i<N.length;i++){
        N[i].cores ← N[i].cores/2;
        N.add(N[i]);
    }

    for(i=0;i < N.length;i++){
        x ← N[i];
        y ← popMatchFromTheEnd(C,L,LC,N);
        if(y != null)  coSchedule(x,y);
    }
    for(i=0;i < LC.length;i++){
        x ← LC[i];
        y ← popMatchFromTheEnd(C,LC,L);
        if(y != null)  coSchedule(x,y);
    }
    for(i=0;i < L.length;i++){
        x ← L[i];
        y ← popMatchFromTheEnd(L);
        if(y != null)  coSchedule(x,y);
    }
    for(i=0;i < C.length;i++){
        x ← C[i];
        y ← popMatchFromTheEnd(C);
        if(y != null)  coSchedule(x,y);
    }
```

The lists of the algorithm contain all the applications of each class and the optimal number of cores that should be allocated to each one. This algorithm iterates over each list of applications and for each application attempts to find a matching application (through method *popMatchFromTheEnd*()), so that the total number of allocated cores do not exceed the available cores of the current package. The function *coSchedule*() increases, if possible, the cores of applications x and y equally, so that the sum is equal to the number of available cores of the system. Note that only for N-class applications, we select to allocate the half cores and schedule the applications 2 times, since their total performance will not degrade due to their cpu-bound profile. It is evident from the algorithm that some matches are attempted to be avoided, because the contention is aggravated. We need to avoid as much as possible the co-execution of L-C, L-L, L-LC pairs of applications, since the interaction between the different sources of contention creates serious performance degradation. The interference of co-execution between all possible combinations of applications is explained in detail in [1]. All the applications that will be left in the end of the algorithm, will execute with all the cores of the package allocated, so that there are no cores left idle. To take full advantage of the cores of the system, the lists of the applications are sorted with the applications requiring the least cores in the beginning and the function *popMatchFromTheEnd*() searches all the lists given as parameters, starting from the end, to find one that can be co-scheduled with x. In this way, it is guaranteed that the cores that will be given in the end to the remaining applications will be the least possible.

A workload has been created with 17 applications from the Polyhedral Benchmark Suite [8], so that we have enough applications from all classes. The whole workload is executed for 1 hour, with quantums of 1 second, as stated before and when an application finishes, it gets respawned to execute again. Our scheduler is compared to LLC-MRB, LBB, the default Linux scheduler and a naive Gang scheduler.

In the ***Gang scheduler***: all available cores (8) are allocated to each application and each application is executed alone.

In the ***default Linux scheduler***: the optimum number of cores as defined from the prediction model are allocated to each application and then the applications are left to be scheduled by CFS. This means that threads of the same applications might not be co-scheduled.

In ***LLC-MRB*** and ***LBB***: the number of allocated cores for each application is defined by the prediction model, but applications are co-scheduled using a sorted list by LLC misses or memory bandwidth correspondingly and combining applications from the beginning of the list with applications from the end.

In our scheduler, called ***LCN***: the number of allocated cores are defined by the prediction model except from N applications, which are allocated the half cores and scheduled twice as described above. The workload is divided into pairs and alone applications using the previous algorithm.

In the following table, we can see the number of times each applications has been completed during the execution of the workload from the different schedulers. The schedulers can be compared in terms of overall throughput using 2 criterias : (i) the total number of times all the applications have been completed (ii) the total number of applications for which the scheduler has managed to achieve the biggest number of completions. The schedulers can also be compared in terms of fairness using the covariance between the completion-times number of each application and the corresponding completion-times number of the Gang scheduler. Green colour is used to point where a scheduler has managed to achieve the most completions for an application and yellow colour is used if there are multiple schedulers that have reached the maximum number. As it is evident, our scheduler outperforms CFS and the other state-of-the-art contention-aware schedulers in terms of overall throughput using both criteria. In terms of fairness, it seems that the other contention-aware schedulers are better with LLC-MRB being the best. However, this is not a clear conclusion, since it may not mean that some applications are unfairly treated by our scheduler, but instead that some applications are highly favoured.

Table 3: *Comparison between schedulers*

Benchmarks	Gang	CFS	LLC-MRB	LBB	LCN
2mm(1024)	54	101	92	92	101
cholesky(1024)	55	124	92	92	136
jacLR	13	15	13	16	17
stream_d0	31	34	32	30	34
stream_d1	31	34	21	32	21
trmm(3000)	54	75	69	69	69
dynprog(700,350)	54	22	36	53	36
mvt(600)	43	31	39	34	30
syr2k(900)	72	80	92	92	92
atax(8000)	27	20	25	19	22
gemver(9000)	54	38	69	69	39
gemm(1150)	72	85	55	55	92
cholesky(4096)	43	49	55	55	55
atax(18000)	43	25	46	21	39
correlation(1500)	43	62	55	55	55
covariance(1500)	43	62	55	55	55
2mm(1500)	72	79	92	92	92
Total exec.	804	936	938	931	985
Most Improved	-	5	7	5	8
Covariance	-	0.46	0.30	0.32	0.46

Investigating the schedulers from a theoretic perspective, we can explain their difference in total throughput. CFS scheduler, being contention-

unaware, cannot locate resource contention and has also the disadvantage that it does not attempt to schedule threads of the same applications together. So, the benefits of the parallel programming are somewhat lost, when threads containing sharing data are scheduled in different quantums. On the other side, both LLC-MRB and LBB cannot differentiate between class N and C applications, since they both exhibit low LLC misses and memory link usage [1]. As a result, they co-schedule class L applications with class C, thus creating more contention. However, our scheduler can tell them apart through the classification scheme and limits the contention even more.

V. Conclusion - Future Work

We implemented a contention-aware scheduler that can be easily incorporated in a real-life scheduling environment. To make this scenario feasible, 2 approaches can be followed. According to the first approach, each application that is inserted in the scheduling queue is initially executed for a really short period (around 2-3 quantums) to acquire the necessary HPC counters for the first 2 steps and then the application is re-inserted in the final queue classified and attached to its prediction. The other approach is to start schedulign the applications with minimal resources and dynamically monitor the HPC counter during the first quantums in order to dynamically adapt their resources afterwards. The dunamic adaptation of the scheduling is attainable, since the control group infrastructure can handle programs that create threads dynamically [10]. After the classification and the definition of resource allocation for each application, the final scheduling algorithm is executed in the current queue to create the optimal combinations.

The granularity of the prediction model can be significantly improved in the future by using stepwise regression models to add more hardware counters into the relationships. Machine learning methods can also be used to improve the accuracy of the prediction model relationships on different architectures. The suggested scheduler is implemented to function in NUMA architectures with only 1 package. However, it can be extended to multiple packages using thread migrations between different memory domains only when beneficial, since similar research has been conducted in the field of contention-aware thread migrations[11].

References

[1] A.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,pages 469-470, New York, NY, USA, 2010. ACM.

[2] S.Blagodurov, S.Zhuravlev, A.Fedorova. Contention-Aware Scheduling on Multicore Systems, Journal ACM Transactions on Computer Systems (TOCS), Article No. 8, New York, NY, USA, 2010, ACM.

[3] Xiao Zhang, Sandhya Dwarkadas, Kai Shen. Towards practical page coloring-based multicore cache management. In Proceedings of the 4th ACM European conference on Computer systems, pages 89-102, New York, NY, USA, 2009, ACM.

[4] Moinuddin K. Qureshi, Yale N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to

Partition Shared Caches. In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, pages 423-432, Washington, DC, USA, 2006, IEEE Computer Society Washington.

[5] Wm. A. Wulf, Sally A. McKee. Hitting the memory wall: implications of the obvious. ACM SIGARCH Computer Architecture News, pages 20-24, Volume 23 Issue 1, March 1995, New York, NY, USA, 1995, ACM

[6] E. Ebrahimi, O. Mutlu, and Y. N. Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In HPCA-15, 2009.

[7] Stream benchmark. https://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/stream.

[8] The Polyhedral Benchmark suite. http://web.cse.ohio-state.edu/~pouchet/software/polybench.

[9] Intel Performance Counter Monitor - A better way to measure CPU utilization. http://software.intel.com/en-us/articles/intel-performance-counter-monitor.

[10] Cpusets: Processor and Memory Placement for Linux 2.6 kernel based systems. http://oss.sgi.com/projects/cpusets/.

[11] The freezer subsystem. https://www.kernel.org/doc/Documentation/cgroups/freezer-subsystem.txt.

[12] Kishore Kumar Pusukuri, David Vengerov, Alexandra Fedorova, Vana Kalogeraki. FFACT: a framework for adaptive contention-aware thread migrations. In Proceedings of the 8th ACM International Conference on Computing Frontiers, Article No. 35, New York, NY, USA 2011, ACM.

[13] Major Bhadauria and Sally A. McKee. An approach to resource-aware co-scheduling for CMPs. In Proceedings of the 24th ACM International Conference on Supercomputing, ICS '10, pages 188-199, New York, NY, USA, 2010. ACM.

[14] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. Contention-aware scheduling on multicore systems. ACM Trans. Comput. Syst., 28(4):8:1-8:45, December 2010.

[15] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In Proceedings of the 5th European conference on Computer systems, EuroSys '10, pages 153-1666, New York, NY, USA, 2010. ACM.

[16] Yuejian Xie and Gabriel Loh. Dynamic classification of program memory behaviors in CMPs. In Proceedings of the 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects, 2008.

[17] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In International Symposium on High Performance Computer Architecture, pages 367âĂŞ378, 2008.