A Resource-Centric Application Classification Approach

Alexandros-Herodotos Haritatos
School of ECE, NTUA
aharit@cslab.ece.ntua.gr

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

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

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

ABSTRACT

In this paper we present a resource-centric application classification approach that monitors data flow along the path from main memory to the cores to locate spots of high resource utilization and potential resource contention. We designate three application classes, i.e. streaming applications, last-level cache sensitive applications and applications that restrict their activity either within the cores or in the private levels of the memory hierarchy. Our classification scheme can form the basis for a number of preliminary prediction models that are capable of predicting application interference with high accuracy.

Keywords

Application classification, contention-aware, prediction

1. INTRODUCTION

Chip Multiprocessors (CMPs) encapsulate several cores that share a number of critical resources such as memory links, cache memory and memory controllers. Applications running simultaneously may compete for these resources, leading to resource contention and eventually to performance degradation. Contention on shared resources may even victimize the threads of a single parallel application moderating, or even mitigating benefits of parallel execution. Beyond performance degradation that can be severe in many cases, performance instability is another critical issue especially in computing environments where performance guarantees need to be maintained. To address the problems created by resource contention, researchers have proposed modifications in hardware (e.g. cache partitioning [12]) or software (e.g. contention-aware scheduling [3, 10, 14]). In all cases, a classification scheme is employed that utilizes information regarding shared resource utilization [2, 3], application resource footprints [6, 7, 15] or co-execution behavior [4, 8].

COSH 2016 Jan 19, 2016, Prague, CZ

DOI: 10.14459/2016md1286948

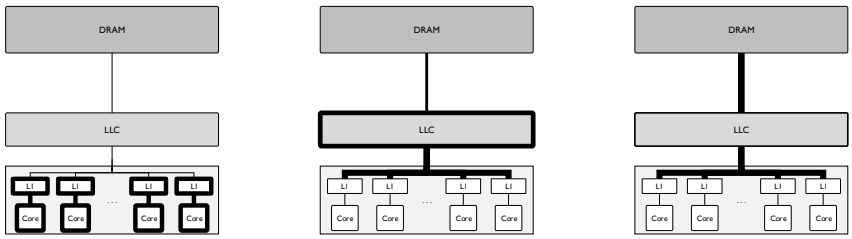
Contention-mitigating mechanisms build upon this knowledge to take better optimization decisions. For example, contention-aware schedulers rely on application classification that predicts interference of co-execution scenarios. Cache utilization patterns [6, 7, 15], LLC miss rate [3], memory link bandwidth [2, 10], contentiousness and sensitivity [14] have been proposed towards this direction. Ultimately, these schedulers aim at reversing the effects of contention on QoS, throughput [16] or energy consumption [10].

The accuracy of the classification scheme in the prediction of application co-execution penalties is one of the most critical factors for a co-scheduling framework. However, most of these schemes capture applications' activity in a limited part of the architecture, i.e. either memory link or last level cache (LLC). Thus, they cannot infer application utilization at each specific hardware resource. In this paper we present a classification scheme based on previous work [5] that inspects the entire memory hierarchy from main memory down to the compute cores and captures data flow and resource utilization. This information is utilized to understand application behavior and predict interference problems. In this way we are able to spot contention on both memory link and LLC. Our classifier distinguishes between three application classes: streaming applications; cache intensive applications; and applications that exhibit no significant activity on the shared resources of the system. We demonstrate interactions between applications from various classes are adequately predictable through simple prediction models, therefore could be applied in an optimized scheduling mechanism.

The rest of the paper is organized as follows: Section II presents our classification approach and Section III presents experimental results. In Section IV we discuss related work. Finally, Section V concludes the paper and discusses ideas for future work.

2. CLASSIFICATION

Our application characterization approach has the following objectives: a) to be capable of locating contention on both the shared memory link and LLC simultaneously, in order to more accurately capture the application's resource utilization pattern, b) to rely solely on information that can be collected at runtime from the existing monitoring facilities of modern processors (requiring no additional hardware support) in order to increase its applicability as much as possible, and c) to be sufficiently fast in order to be capable of supporting prompt decisions, required in dynamic execu-



(a) class N (b) class C (c) class S

Figure 1: Activity in application classes

tion environments where applications enter, exit and change behavior frequently.

2.1 Application classes

In our analysis, the following three application classes are relevant:

Class N: Applications that restrict their activity either within the core or in the private caches of the core. The members of this class create no contention to the shared system resources. The class includes applications with heavy computations, very small working sets or optimized data reuse that can be serviced by the private caches.

Class C: Applications with high activity on the shared LLC. This is a wide class including members with a combination of main memory access and LLC data reuse, or members with varying characteristics, such as those that operate on small data sets with heavy reuse, optimized code for the LLC (e.g. via cache blocking with a block size fitting the LLC), or latency-bound applications that make irregular data accesses and benefit a lot from LLC hits.

Class S: Applications of this class have a stable data flow on all links of the memory hierarchy. This class typically includes applications that perform streaming memory accesses on data sets that largely exceed the size of the LLC, or have either no reuse or large reuse distances Although they fetch data on the entire space of the LLC, they do not actually reuse them either because their access pattern does not recur to the same data, or because they have been swept out of the cache. No level of cache memories helps S applications accelerate their execution. Thus, they tend to pollute all levels of caches. Figure 1 indicates the activity spot of each class.

2.2 Classification method

Having defined the application classes, we need a concrete method to perform the classification using runtime statistics. The core idea is to inspect the data path from main memory down to the core to locate links with high utilization. We have focused only on the stream flowing towards the core, as we have empirically found that this direction concentrates the largest portion of contention. Figure 2 illustrates this idea.

Our classification method implements the decision tree shown in Figure 3. We follow a hierarchical approach in the classification. First, we look at the application activity in L1 cache. No activity in L1 means that data used from the entire path of memory are too few, indicating that the application's activity is restrained within the core. Applica-

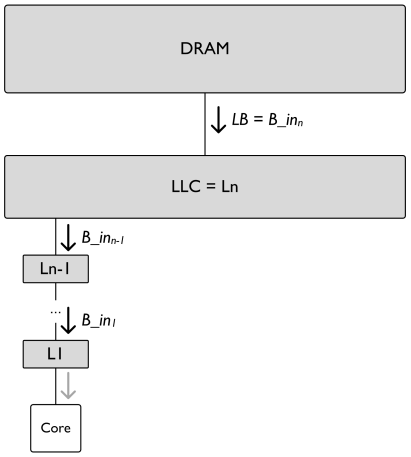


Figure 2: Inspected data flow in the memory hierarchy

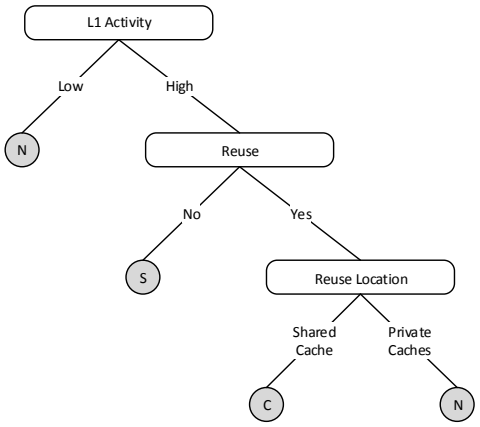


Figure 3: Decision tree for application classification

tions that exhibit this attribute are classified as N. If we are unable to locate reuse at any level of the cache hierarchy, then the application has a streaming attitude and is marked as S. As cache reuse factor, we use the ratio $CR_i = \frac{B_{in_{i-1}}}{B_{in_i}}$. The rationale is simple: if data flow out of a cache towards the core with a much higher rate than they flow in, then we can safely assume that reuse is present. We empirically set a threshold of 2 to designate reuse. If there is cache reuse, reuse location needs to be examined. If reuse is higher in the private caches, then the application is classified as N,

else the application is classified as C (as the dominant reuse is on the LLC).

2.3 Co-execution effects

Despite the fact that inside each class one may find applications with quite different execution patterns, the classes themselves can be used to capture the big picture of how applications access common resources and of co-execution interference between applications. In the following we denote xy the co-execution of an application from class x with an application from class y. We use $*$ as a wildcard for "any class". Here is what we expect from the co-execution of all combinations:

$N - *$: As applications from class N do not activate in any shared resource, this co-execution does not create interference with any of the applications.

$S - S$: Applications compete for the memory link. The contention pattern in this case indicates that the shared resource which in this case is the memory link bandwidth is divided (not necessarily equally) between the competing applications.

$S - C$: As S applications tend to pollute caches, C applications may suffer from the coexistence with S applications. In cases of high pace of data fetching their co-execution can be catastrophic for the C application. On the contrary, S applications having low pace of data fetching may cause no harm to C applications. The streaming nature of S applications causes data that are potentially heavily reused by C applications to be swept out of the LLC rapidly, enforcing them to access main memory. This contention pattern can lead to dramatic slowdowns for C applications. On the other hand, S applications suffer no severe penalty from co-execution.

$C - C$: This is the most difficult to predict co-execution. In the general case, we expect cache organization and replacement policies to be able to handle adequately high activity from different applications on the shared LLC. However, if both exhibit similar data access patterns, contention is expected to be high. To go deeper into the class and better understand possible interactions, one would require information on the data allocated to each application on the LLC and the access pattern. This would require either information from static analysis, or additional hardware support.

2.4 Interference prediction models

A successful classification model is the core of an accurate prediction model that will ultimately be used by resource management mechanisms such as contention-aware schedulers. Interference prediction is the most useful one, since it can be directly applied to scheduling policies. Interference effects are more complicated as the number of applications running simultaneously, increases. To confirm the validity of the expected co-execution effects, co-execution scenarios have to be simple. Therefore these scenarios consist of two applications competing for shared resources of the same package at the same time. There are six possible scenarios derived from our classification model, described next. We denote D_{X-Y} the degradation of application X when executed with application Y.

$S - S$: In this case both applications compete for the memory link. As the available bandwidth of the link is limited, applications need to share it. Even S applications that are running alone may have a small LLC reuse, this will be lost due to the high pace of LLC pollution, that their competitor causes. Because of this, we expect that the degradation will be expressed as:

$$D_{S-S} = \frac{BW_{max}}{BW^{in}_{mem \to LLC} + BW^{co}_{mem \to LLC}}$$

where BW is the bandwidth, in denotes the inspected application and co the co-executed one.

$S - C$: S applications invalidate C applications' cached data. We expect that as the bandwidth of S applications increases, the consequences will be more dramatic for C applications. The wide pallet of different memory access patterns demands observation on a large number of parameters. However, as an initial attempt, we experiment with a linear function of degradation with the bandwidth of the co-executed application. Thus, our prediction model is expressed as:

$$D_{C-S} = \alpha \times BW^{co}_{mem->LLC} + \beta$$

On the other hand, we expect S applications to suffer minimal pain, and we set $D_{S-C} = c_1$, with c_1 taking a value close to 1.

$C - C$: As cache hardware is expected to efficiently handle the conflicts in this case, degradation prediction is defined as a constant value taken from the average value of our experimental results, i.e. $D_{C-C} = c_2$

$N - S$, $N - C$, $N - N$: N applications do not affect and are not affected by other applications. In all of these cases all the degradation's predictions are taken as constants, i.e. $D_{S-N} = c_3$, $D_{C-N} = c_4$ and $D_N = c_5$, again with c_3, c_4 and c_5 taking values close to 1.

We calculate parameters α, β and $c_1 \dots c_5$ with regression utilizing experimental data from the co-execution of our application test suite, which is presented in Section 3.2. We split our set in sliding windows of 80% training and 20% testing sets.

3. EVALUATION

3.1 Experimental Platform

The evaluation of the classification scheme is performed on an Intel® Xeon® CPU E5-4620. The architectural details are presented in Table 1. All the available hardware prefetchers are enabled whereas Hyperthreading and Turbo Boost are disabled. The platform runs Debian Linux 6.0.6 with kernel 3.7.10.

Cores	8
L1	Data cache: private, 32 KB, 8-way, 64 bytes block size Instruction cache: private, 32 KB, 8-way, 64 bytes block size
L2	private, 256 KB, 8-way, 64 bytes block size
L3	shared, 16 MB, 16-way, 64 bytes block size
Memory	64 GB, DDR3, 1333 MHz

Table 1: Processor details

In order to monitor the applications and acquire their profile, we employ hardware performance counters to collect performance data. Specifically, we use `UNHLT_CORE_CYCLES, INSTR_RETIRED, L1D.REPLACEMENT, L2_LINES.IN`. Furthermore, we use `OFFCORE_REQUESTS (0xB7, 0x01; 0xBB, 0x01)`.

3.2 Experimental Setup

In our evaluation, we populated all four application classes selecting benchmarks from a variety of multithreaded suites

Name	Source	Details	$B_{in_{3=LLC}}$ (GB/sec)	$CR_{llc} = \frac{B_{in_2}}{B_{in_3}}$	$CR_{l2} = \frac{B_{in_1}}{B_{in_2}}$	IPC	class
jacobi_l	polybench	large data set	10.321	1.50	1.02	0.559	S
stream	[9]	array size = 50000000, offset = 0, NTIMES = 10, kernel = TRIAD	11.646	1.00	1.03	0.724	
bt	NAS	class A	8.566	0.97	1.36	0.813	
fw	polybench	small data set	7.468	1.02	1.16	1.903	
Dynprog	polybench	custom data set	8.220	0.98	1.54	1.384	
Mvt	polybench	custom data set 1000	0.006	4.21	565.29	0.455	C
Mvt	polybench	custom data set 6000	2.274	0.92	8.31	0.343	
atax	polybench	large data set	2.426	0.82	8.61	0.336	
cg	NAS	class B	5.955	1.37	4.38	0.728	
syr2k	polybench	large data set	5.883	2.07	2.00	1.919	
gemver	polybench	large data set	2.873	0.82	7.23	0.440	
cholesky	polybench	large data set	0.156	0.85	195.56	1.876	
pchase	[1]	A pointer-chasing benchmark with data set fitting in LLC.	0.135	1.00	222.01	0.148	
lu_t	inhouse	Classic implementation of tiled LU decomposition with data set fitting in LLC.	0.009	8.67	178.91	1.635	
jacobi_s	polybench	small data set	0.534	1.15	35.17	1.209	
fw_t	inhouse	Implementation of tiled Floyd-Warshall algorithm from [13]. Data size fits in LLC.	0.063	9.92	34.45	2.181	
ep	NAS	class A	0.000	0.97	2644.41	0.746	N
mm_t	inhouse	Classic implementation of tiled Matrix Multiplication with data set fitting in LLC.	0.030	17.82	5.68	2.933	
blackscholes	PARSEC	large data set	0.186	0.65	5.63	1.732	

Table 2: Application suite

and inhouse implementations of classic kernels. Table 2 presents our application suite, together with key metrics used to classify each application according to our scheme. The test platform includes one shared LLC and two private caches (L_1 and L_2). In order to evaluate our classification scheme, we co-execute all possible pairs of applications, totally 361 pairs. We allocated half of the available cores to each application. If an application terminates, it is respawned in order to keep contention at the same level.

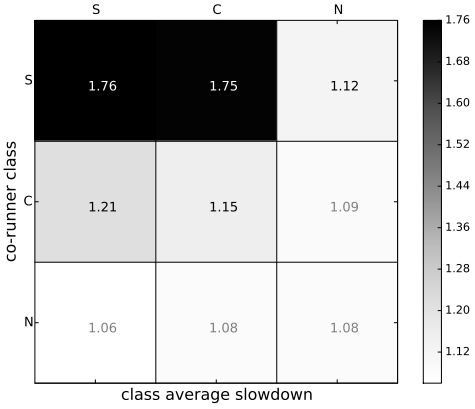


Figure 4: Average application slowdown due to co-execution at the class level. The horizontal dimension represents the slowdown imposed by each class, while the vertical dimension shows the slowdown suffered.

Figure 4 provides a class-level view of average slowdowns of each class measured. We may observe that our classification scheme is able to capture the general trend in interference penalties. In particular, we observe that whenever a N application is involved, the interference overhead is negligible. On the other side of the spectrum, the most contentious pairings arise in S-S and S-C collocations.

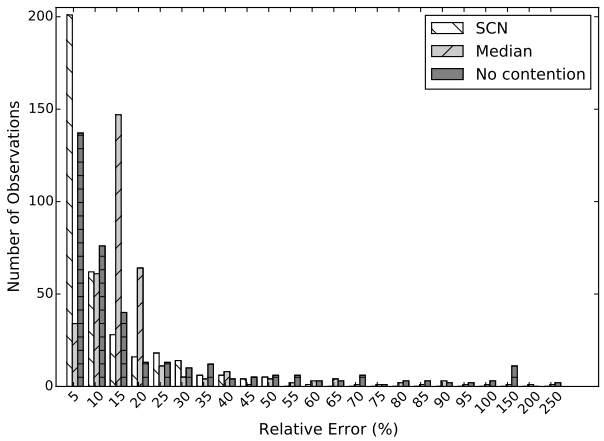


Figure 5: Relative errors for the SCN, 'median' and 'no contention' predictors.

Figure 5 shows the absolute values of the relative errors for all co-executions for our predictor (SCN) and two naive predictors, i.e. the 'median' predictor that always predicts the median of all cases (1.27) the 'no contention' predictor that neglects the effects of contention and always predicts a degradation factor of 1.05. Note that we favored the naive predictors by using the same training and testing sets (i.e. we did not split training and testing sets in 80% and 20% subsets as we did for SCN). We may notice that SCN achieved a prediction error of less than 5% for 55% of the co-executions and less than 30% for 94% of the co-executions. On the other hand, the naive predictors provided guesses

with significantly lower accuracies, i.e. the 'median' predictor had a prediction error of less than 5% for 9% of the co-executions and less than 30% for 89% of the co-executions, and the 'no interference' predictor had a prediction error of less than 5% for 38% of the co-executions and less than 30% for 80% of the co-executions.

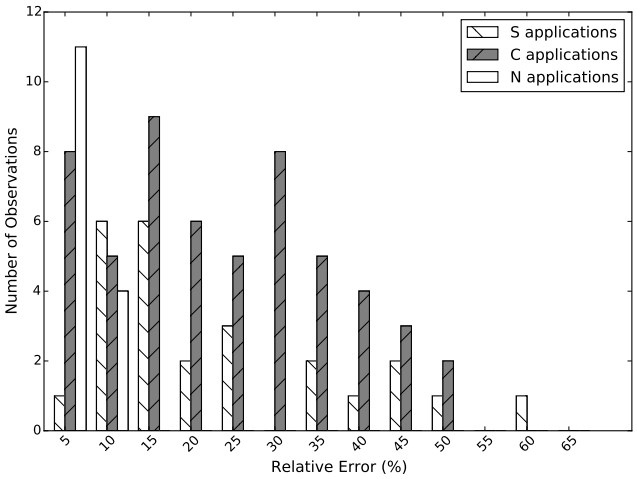


Figure 6: Co-execution with S.

Figure 6 shows the absolute values of the relative errors of SCN prediction for applications of all classes when they are co-executed simultaneously with S class applications. We may notice that despite their simplicity, the prediction models for C–S and S–S co-executions are able to capture the general degradation penalties and keep the prediction errors lower than 30% for 72% of the co-executions. On the other hand, as expected co-execution of S with N applications is predicted with high accuracy.

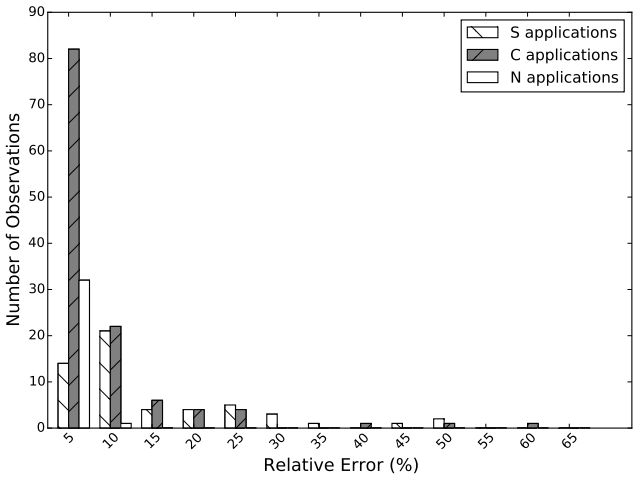


Figure 7: Co-execution with C.

Figure 7 shows the absolute values of relative errors of SCN prediction of all classes when executed with applications from the C class. Quite interestingly, our simple constant predictor for the C–C co-execution keeps errors lower than 30% for 97% of the co-executions, showing that in our experimental scenarios cache sensitive applications tend to share gracefully the shared LLC cache. We also note that S and N co-executions with C applications are accurately

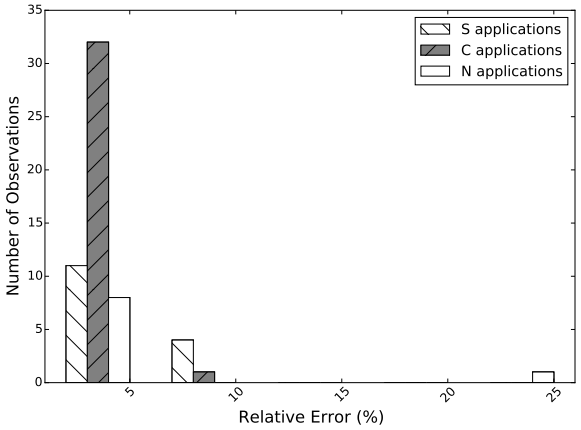


Figure 8: Co-execution with N.

predicted. Finally, Figure 8 shows the absolute values of relative errors of our predictor for applications of all classes when they are co-executed simultaneously with N class applications. Clearly, this is an easy to predict scenario since N applications indeed cause no substantial harm to the their co-executors, a fact that is captured by our predictor.

4. RELATED WORK

Various characterizations schemes have been proposed in the past. Lin et al. [7] proposed a scheme that partitions the LLC cache between two programs using cache coloring. The scheme is employing a program classification based on each program's performance degradation when running using a 1MB cache compared to running using a 4MB cache. Looking also at cache partitioning, Moreto et al. [11] proposed two metrics, namely $w_{P\%}$ and $w_{LRU(th_i)}$. The first metric refers to the number of ways needed by a benchmark to reach at least the P% of its maximum IPC, while the second one refers to the ways allocated to each thread by LRU when two benchmarks are executed together. Based on these two metrics applications were classified in one of three different classes.

Xie and Loh [15] proposed an animalistic approach of the application classification problem. All applications may belong to one of four classes, named Turtle, Sheep, Rabbits and Tasmanian Devils. Applications that do not make much use of the LLC are turtles, while applications that make use of the LLC but are not sensitive to the number of ways allocated to them belong to the sheep group. Rabbits are applications that are very sensitive to the ways allocated to them, and, finally, devils are the applications that make use of the LLC while having very high miss rates.

Jaleel et al. [6] categorize applications in four classes. Core Cache Fitting (CCF) are applications with a dataset that fits in the lower levels of the memory hierarchy and do not benefit from the shared level of cache. On the other hand, LLC Thrashing (LLCT) applications have a data set bigger than the available LLC. Under LRU, these applications degrade performance of any co-running application that uses the LLC. LLC Fitting (LLCF) applications require almost the whole LLC and their performance is severely impacted if there cache trashing occurs. Finally, LLC Friendly (LLCFR) applications are ones that even though they could improve their performance using more cache resources, they do not

suffer significantly when these resources are reduced.

Finally, Tang et al. [14] are employing two metrics for each application, namely Contentiousness and Sensitivity. Based on these, applications belong to one of the following four classes: 1) contentious and sensitive; 2) not contentious and insensitive; 3) contentious but not highly sensitive; or 4) not highly contentious but sensitive. Contentiousness and sensitivity are statistical numbers but, as authors claims, can be easily calculated through performance counters.

5. CONCLUSIONS AND FUTURE WORK

In this paper we presented a resource-centric application classification scheme that monitors data traffic across the entire memory hierarchy, using existing hardware monitoring mechanisms. We base a number of preliminary interference predictors on the classification scheme and evaluate them on a set of parallel applications. The prediction accuracy is very promising and sets a solid basis to support contention-mitigating mechanisms. As a future work, we intend to extend this work in the following directions: a) augment our prediction approach to minimize errors and mispredictions, b) apply this model in scenarios where more than two applications run concurrently c) apply this work to a contention-aware application scheduling environment.

6. ACKNOWLEDGMENTS

This research was funded by project I-PARTS: Integrating Parallel Run-Time Systems for Efficient Resource Allocation in Multicore Systems (code 2504) of Action ARISTEIA, co-financed by the European Union (European Social Fund) and Hellenic national funds through the Operational Program Education and Lifelong Learning (NSRF 2007-2013).

7. REFERENCES

[1] pChase benchmark. https://github.com/maleadt/pChase.

[2] 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 189–199, New York, NY, USA, 2010. ACM.

[3] 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.

[4] Christina Delimitrou and Christos 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.

[5] Alexandros-Herodotos Haritatos, Georgios Goumas, Nikos Anastopoulos, Konstantinos Nikas, Kornilios Kourtis, and Nectarios 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.

[6] Aamer Jaleel, Hashem H. Najaf-abadi, Samantika Subramaniam, Simon C. Steely, and Joel 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.

[7] 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 367–378, 2008.

[8] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *MICRO*, pages 248–259, 2011.

[9] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[10] 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 153–166, New York, NY, USA, 2010. ACM.

[11] Miquel Moretó, Francisco J. Cazorla, Alex Ramírez, and Mateo Valero. Explaining dynamic cache partitioning speed ups. *IEEE Computer Architecture Letters*, 6(1):1–4, 2007.

[12] Konstantinos Nikas, Matthew Horsnell, and Jim D. Garside. An adaptive bloom filter cache partitioning scheme for multicore architectures. In *ICSAMOS*, pages 25–32, 2008.

[13] Joon-Sang Park, Michael Penner, and Viktor K. Prasanna. Optimizing graph algorithms for improved cache performance. *IEEE Transactions on Parallel Distributed Systems*, 15(9):769–782, 2004.

[14] Lingjia Tang, Jason Mars, and Mary Lou 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.

[15] 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.

[16] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys*, 45(1):4:1–4:28, December 2012.