An Overview of Scheduling Policies Targeted at Improving Performance/Efficiency in Multi-cores

Saurav Kumar
North Carolina State University

ABSTRACT

With the need for higher performance, and barriers such as ILP wall, Memory wall and Power wall, multi-core computers have found their way into wide variety of applications in both user and commercial domain. They offer more computing resources while keeping the power considerations in manageable limits. A wide variety of implementations exist in multi-core architecture include SMP (Symmetric multi-processors) and AMP (Asymmetric multi-processors) with heterogeneous systems as a subset of AMP. Multiple challenges have cropped up due to the shift to concurrent programming and widespread adoption of these systems. These include shared resource contention, performance challenges, energy efficiency, fairness and throughput. This paper compares and contrasts recent solutions which have been proposed to address these problems. The solutions which have been taken into consideration include scheduling and related classification schemes, hardware/software partitioning, PMU (performance monitoring unit) assisted scheduling and work-stealing.

Categories and Subject Descriptors
D.4.1 [Operating Systems]: Process Management—Scheduling

General Terms
Algorithms, Management, Measurement, Performance, Design, Reliability.

Keywords
Multicore processors, shared resource contention, scheduling, Symmetric multicore, Asymmetric multicore, frequency scaling, fairness.

1. INTRODUCTION

Multi-cores have long existed, ever since mid-1960s. However due to the complexity involved in multi-core programming they were largely used by the skilled engineers and scientists. Their absence in mainstream systems and the increase in single core complexity and component density based on Moore's law made multi-core systems expensive. Hence, they had a limited customer base usually targeted at batch processing.

* Last updated n March 2013

With the decline in transistor scaling and the hardware development hitting ILP wall, memory wall and power wall, there was a paradigm shift to multi-core computers. Multi-core offered reduced power consumption for perfectly parallelizable workloads. However, achieving ideal speedup is affected by various factors. Based on the target hardware these solution have been classified into two categories: Solutions for SMP systems and solutions for AMP systems.

For SMP systems scheduling is proposed as one of the solutions because it needs software changes, hence can be easily integrated into current kernels. Implementations of new scheduling policies which targets shared resource contention requires two steps: First is identification of classification schemes for threads to indicate what effect they have on each other due to shared resource contention when co-scheduled. Here shared resource includes cache, memory controller, memory bus and prefetching hardware. Second is to devise a scheduling algorithm using one of the classification schemes and demonstrate its performance ("DI performs within 2% of optimal") [1]. Scheduling policies also have effective impact at improving QoS (Quality of Service). Another policy which aims at reducing the energy delay product utilizes task activity vectors to classify applications based on resource utilization. Accordingly, applications using complementary resources are selected and frequency scaling is used if scheduling is not able to properly manage the resource contention.

Efficient HW SW partitioning technique has also been proposed using scheduling and mapping algorithms. On similar lines combination of both hardware and software has been employed in solving the shared cache contention problem in multi-core systems. The hardware can use modified replacement policies so as to allocate cache on need basis for the applications. Since co-scheduling decisions rely greatly on the underlying shared cache replacement policies, the hardware/software approach is a nice avenue for improvement. Balanced work stealing (BWS) is also among one of the novel solutions which utilizes a work-stealing scheduler for improving performance and fairness using two methods. Firstly, it tries to balance the cost and benefits of threads which are awake where costs are resources consumed when an attempt to steal is made and benefits are the available tasks that get stolen by the requesting thread.

Secondly, while making a steal attempt/request the core being used by the attempting thread can be borrowed by another working thread so that the core is effectively retained by the application yet is used for performing work. BWS compared with another work-stealing scheduler Cilk++ has been found to deliver increased throughput by 12.5% and it reduces unfairness from 124% to 20% [6].

In case of AMP systems, the faster cores are usually used to speed up sequential phases of applications and the slower cores allow gaining energy efficiency for execution of the parallel phases. Heterogeneous processor cores can be used with lottery based scheduling policies to provide better performance and

energy savings over state-of-the-art heterogeneous-aware scheduling techniques. This technique focuses mainly on energy efficiency. CAMP or the comprehensive scheduler for AMP aims at increasing both energy efficiency and throughput/parallelism.

The remainder of this paper is classified into following sections: (2) SMP systems and scheduling policies aimed at solving problems in these systems, (3) AMP systems and corresponding scheduling solutions, (4) Comparison of different techniques, (5) Acknowledgements and (6) References.

2. SMP SYSTEMS

Various scheduling policies aimed at reducing contention, increasing performance and energy efficiency are discussed below

2.1. Distributed Intensity Scheduler (DI)

Majority of scheduling policies currently aim at ensuring fairness in terms of CPU Time and minimizing the period for which individual cores are idle. However, contention for shared resources can greatly affect the execution time of individual processes. In case of a shared lower-level cache, performance can be improved if a better scheduling policy is selected, by as much as up to 50%.

2.1.1. Classification schemes

The algorithm proposed by Jiang et al. [14] was used as a reference to compare the performance of classification schemes. This algorithm takes con-run degradations as one of the input and is guaranteed to find an optimal scheduling assignment. In order to evaluate the classifications schemes, an optimal schedule and an optimal classification scheme were identified using Jiang's method. The average performance degradation for this setup was recorded. Then an estimated best schedule was determined using Jiang's method and the classification scheme to be evaluated. The difference between average performance degradation of optimal and the evaluated scheme gave an idea of how the evaluated scheme performs. The smaller the difference, the better is the performance of the evaluated scheme.

Four classification schemes were studied which use information provided in stack distance profile: Stack Distance Competition (SDC) [11], Animal Classes [12], Solo Miss Rate [13], and the Pain Metric [1]. A stack distance profile provides information about the behavior of an application with respect to cache line reuse pattern.

a. SDC: - SDC classification scheme models how two applications affect each other's miss rate while competing for LRU stack positions. Performance degradation can be derived from this information.

b. Animal classes: - based on mapping the application to four different classes i.e. turtle (shared cache used rarely), sheep (has low miss rate, insensitive to the cache associativity allocated), rabbit (low miss rate, sensitive to allocated cache associativity) and devil (high miss rate, does cache thrashing) the degradation when two classes are co scheduled is rated between numbers 0 and 8. Then the combination with lowest result is selected and scheduled.

c. Miss rate: Applications with highest miss rate are scheduled in different caches so that they do not affect each other. Since, high miss rate implies contention at memory controller, memory bus and prefetching hardware.

d. Pain classification scheme: Two parameters i.e. cache sensitivity and cache insensitivity are combined to produce Pain metric. Sensitivity is how much cache space contention affects an application. Insensitivity is how much an application affect others by taking the cache space allocated in shared cache.

2.1.2. Scheduling policies and observations

A cache unaware scheduler might end up picking good and bad assignments of processes to individual cores based on their respective probabilities. However, contention-aware scheduler can deliver QoS, since the algorithm ensures that a bad scheduling assignment is not made.

As cores in a multi-processor increase, the probability with which imperfect classification schemes get a good mapping decreases. Hence, a good classification scheme is a pre-requirement for reducing performance degradation in multi-core system. Parameters affecting performance degradation include contention for shared cache, DRAM controller, front side bus and prefetching hardware. The Miss rate takes into account all of these and is easiest among the 4 schemes to implement, hence was selected for further study. Distributed Intensity algorithm (DI) based on centralized sort policy which maps threads to cores based was used along with the Solo LLC miss rate to compare performance with current Linux scheduling algorithms

The performance of the new scheduling policies when compared with default contention-unaware Linux scheduler showed that the worst-case performance of individual applications gets reduced (by up to 13%) and has lesser variance in competition times. This enables the contention-aware scheduling to provide QoS and performance isolation.

2.2. Scheduling Aimed at Energy Efficiency

Contention for resources affects both completion time of programs and energy efficiency of the system. In order to avoid hardware complexities, frequency and voltage selection across all cores is majority of systems is kept same. However, frequency and voltage are usually determined based on the program in order to provide energy efficiency. In case of multi-core systems, depending on the combinations of processes running of different cores, not all of them run under optimal frequency. Hence, a scheduler can be used to alleviate this problem. Tasks with different characteristics can reduce resource contention, on the other hand tasks with similar characteristics cater themselves to be easily scheduled under optimal frequency.

Tests of AMD Opteron 2354 quad-core chip and Core2 Quad Q6600 suggested that when memory bound tasks are co-scheduled then no major benefit is gained by scaling the frequency for optimal operating conditions. For further analysis the paper takes EDP (energy delay product i.e. the product of energy spent by the processor for a certain task multiplied by its total runtime) as a metric of comparison. A run of four SPEC CPU 2006 micro benchmarks on Core 2 Quad showed that memory bandwidth turns out to be a critical resource when it comes to contention. Tests indicate that combining tasks in order to reduce resource contention is more important than combining tasks that share a common optimal frequency.

In order to analyze resource utilization, task activity vector is used where activity vector quantifies the degree at which various resources are used by a particular task running at max frequency. The values is this vector range between 0 (no utilization) and 1 (complete utilization). The performance monitoring unit of processors provides information about the values for this vector.

Resources considered include memory bus, L2 cache and rest of the core (resources not shared between cores) A translation vector is used to compare the activity vectors recorded at different frequencies.

Based on the activity vector information Vector balancing method is proposed for reducing resource contention. The aim is to have multiple tasks with different characteristics on each core so that the scheduling policy can pick suitable tasks for each course. In order to get high variance in the run-queues task migrations are performed. Cross node (VM based) migration is also an addition, however it will not be considered since it is out of scope for this paper. The run queue is kept sorted lazily with low overhead and the cores are grouped into pairs of two so as to allow executing tasks with complementary resource demands. This scheme can be in conflict with schemes applied for I/O bound tasks, hence is not feasible for I/O intensive workloads. In situations where the scheduling policy cannot reduce contention, frequency scaling is employed to reduce EDP.

The final deployment on a Linux system of the above mentioned strategies showed that the activity vector calculations did not have any effect on total runtime for the SPEC benchmarks. Additionally, co-scheduling tasks which have complementary resource requirement reduces contention significantly.

2.3. Policies Requiring Hardware Support

2.3.1. Effective Hardware Software Partitioning

The Decisions regarding partitioning of a system into hardware and software directly affect cost/performance characteristics of the final design. A partitioning scheme can be used to reduce total CPU time taken by a process.

Different tasks are categorized into nodes where each node is a set of instructions which should be executed sequentially without pre-emption. Dependencies are also taken into account to generate a task precedence graph. A combination of scheduling algorithm and converting tasks taking longest time to be implemented in hardware instead of software (using FPGA) is used to provide speedup. This also allows in identifying best architecture for a given application.

2.3.2. Cache replacement and Utility aware scheduling

Intelligent software scheduling approaches including the ones which take miss rate from CPU performance measurement units into consideration are one way of solving the LLC contention problem. Another way for serving applications better is to monitor the heterogeneity in demand for cache and use smart replacement techniques in hardware to allocate proportional resources. A combined approach can be explored for obtaining better performance.

LRU based replacement has been shown to perform poorly when it comes to shared caches. To avoid worst application schedule, majority of the intelligent scheduling policies avoid co-scheduling CPU bound application with memory bound application. However, these studies use an LRU managed LLC. When the LRU policy is replaced with smart cache replacement policies, the performance variations for workloads reduce (<4%) hence the benefit of using smart scheduling policies is not significant.

Effectively the smart replacement policies minimize burden on software for reducing resource contention and increasing performance. As compared to other policies which use colors, animals or miss rates to classify applications according to their memory intensity, CRUISE uses four new classes:

CCF (Core cache Fitting): applications which have a working set that fits in their private cache.

LLCT (LLC thrashing): Similar to streaming applications which have huge working sets

LLCF (LLC fitting): Need majority of the shared LLC and the performance reduces as LLC allotment is decreased.

LLCFR (LLC friendly): They perform well as the LLC allotted is increased however performance degradation is not significant even if a major chunk of LLC is not allotted to them.

In order to gather information regarding application cache utility, the applications can be periodically paused on a CMP to gather information about cache use. However, this policy incurs extra performance overhead. Hence a runtime isolated cache estimator (RICE) is proposed which monitors cache utilization per application based on Set Dueling Monitor (SDM). The SDM dedicates few sets of cache to follow a certain policy, and then uses the resultant cache misses. Hence, two SDMs are dedicated per application and all other applications bypass these cache sets.

RICE however needs modification in the ISA to enable software control. Using this information, a scheduling policy can be devised based on the underlying LLC replacement policy. For a LRU managed shared LLC the policy follows following rules (CRUISE-L):

a. All LLCT applications are mapped on same LLC

b. CCF applications are distributed across LLCs

c. The LLCF applications are co-scheduled along with CCF

d. LLCFR applications are mapped to remaining places.

In case of DRRIP, a low overhead, high performance shared cache replacement policy the scheduling steps are (CRUISE-D):

a. LLCT applications are distributed across all LLCs

b. CCF applications are also distributed across LLCs

c. LLCF and CCF/LLCT applications are co-scheduled

d. LLCFR applications are filled in remaining slots

When compared with random, Distributed intensity and Worst application schedule policies, CRUISE significantly reduces performance variation. In case when the workload consists of mixes with high performance variation, CRUISE frequently arrives at an optimal schedule. In both LRU and DRRIP managed share caches, the performance variation using CRUISE lies below 5%. Both CRUISE-D and CRUISE-L are within 1% of the optimal assignment schedule (OAS), in case of fairness and the performance is similar to OAS when it comes to throughput and weighted speedup. It proves to be robust across all metrics. The CRUISE policy also proves to be quite scalable as it bridges performance variation between OAS and WAS when a 4-core system is scaled to 8-core system with proportional increase in applications/threads outperforming all other policies taken into consideration. However in case of lower workload, both CRUISE and DI show degraded performance to within 5% when compared with the WAS policy due to the cost of overheads.

The functioning of RICE was validated by reducing the LLC size and analyzing behavior of working sets. It was found that cache fitting or friendly applications were changed to cache thrashing or

fitting when the cache size was reduced and vice versa. Hence, RICE responds to dynamic changes in available LLC size.

CRUISE tackles majority of problems faced by other policies. In case of the DI policy, the authors suggest that the contention for DRAM controllers and hardware pre-fetchers also account for majority of the performance degradation. However, in recent processor designs, the pre-fetchers have been made private and are located at L2 cache and the DRAM controllers are moved on die and are more sophisticated thereby reducing the sharing effects in case of a LRU managed cache. Other techniques like software cache partitioning technique which allocates a certain portion of cache to individual applications is complicated and requires intricate changes in the virtual memory policies used by the kernel. Software-centric algorithms which sample performance data of threads and learn their behavior suffer due to dynamic phase changes and their complexity increases with the increase in number of threads. Hence, CRUISE is the most feasible policy providing scalability and high performance for SMP systems.

2.4. Work-Stealing Based Scheduling Policy

An aspect which comes into play in case of multi-programming is a certain application spawning threads to make use of multiple-cores. These threads are then scheduled based on the user-level task scheduler. The tasks to be performed by the application are distributed across multiple worker threads felicitating the concurrent execution of the application. In such cases work stealing proves to be quite effective when it comes to reducing complexity of parallel programming. However, since an operating system does not recognize the tasks of each of the worker threads, wasteful threads might get scheduled which cause reduced throughput. Additionally, when work stealing schedulers try to alleviate this problem by introducing yielding for the wasteful threads then this yielding causes loss of cores for the application leading to reduced concurrency.

BWS (Balanced work stealing) tackles this issue by putting wasteful threads to sleep based on data derived from OS. It wakes up the threads when they are likely to contribute towards application progress and allows yielding to another thread of same application hence avoiding the application from being blocked for a particular core.

In case of work-stealing a user level scheduler assigns tasks to multiple worker threads. The workers which process and generate tasks are called busy workers. Useful thieves distribute or yield when a parallel task needs the processor. Wasteful thieves on the other hand waste the resources in their successive work-stealing attempts which turn out to be unsuccessful. Time slicing allows core to be assigned to individual applications. However, throughput depends on ability of an application to completely utilize the allocated resources. Static partitioning relies on similar application characteristics. Hence, work stealing can prove to be a powerful tool which allows several applications to run concurrently in a multi-core system.

Wasteful thieves are a major concern for work-stealing algorithms and degrade the performance by 15%-350%. Current work-stealing algorithms like ABP (proposed by Arora, Blu-mofe, and Plaxton), cause unfairness when the thieves which yield are not resumed so as to complete their work in time i.e. when they transition from being wasteful to useful thieves and reduce throughput when the thieves do not yield cores or yield cores but are invoked frequently. The OS scheduling policies also affect this indeterministic behavior of thief threads.

Based on runs of four benchmarks on 32 core machine it was observed that co-scheduling two work stealing applications increases throughput when compared with time-slicing algorithms. However, an unyielding work-stealing scheduler might degrade performance by as much as 600%. Hence in order to aid support of work-stealing schedulers, support has to be provided from the OS which is one of the implementations in BWS.

BWS prioritizes busy workers and thieves yield to the busy workers so that time slice given to a work-stealing application is utilized intensively. The stealing costs are controlled by putting to sleep/waking thieves. In cases where resources are plenty multiple thieves can be woken up to increase efficiency/concurrency of the application. The schedulers acquire running status of workers from the OS and an additional support which enables the core to be transferred to peer workers. With this implementation BWS proves to be better than ABP when it comes to fairness, efficiency and throughput.

BWS uses the thieves to perform management work of waking up worker threads in order to reduce work overhead. Thus when a thief is unsuccessful in its steal attempt, it performs management task so as to have some effective work done (in this case for the BWS algorithm). In order to avoid a case where all thieves are asleep, a 'watchdog' thief is selected which always remains awake and appoints a new watchdog in case it is able to successfully steal resources for a task.

Based on tests performed with multiple benchmarks on a 8-core system, BWS outperformed Cilk++ scheduler (with ABP) in terms of fairness and throughput (11%-33% on average). Due to the well managed wasteful thieves the context switches being incurred by BWS are greatly reduced which in turn reduce s the penalty which is incurred by Cilk++ (reduced by >70%) .

The sleep threshold calibration in BWS is a sensitive parameter since a large value might cause reduced throughput at reduced workloads but may increase fairness at higher workloads. An optimum value for this parameter was found to be between 32 and 256.

When compared with WSGI (work stealing with global information) where the information about all workers (busy and thieves) is maintained in a global repository, BWS proves to be more efficient since local information which is stored in the watchdog and peer thieves reduces contention. Benchmarks are on an average 43% slower with WSGI as compared to BWS.

3. AMP SYSTEMS

Scheduling policies aimed at reducing solving problems in AMP multi-core systems are discussed below

3.1. Lottery Based Scheduling

By using heterogeneous cores with same ISA but different power and performance characteristics, both power and performance benefits can be provided. Other methods employ scheduling decisions based on thread bias which is computer either using computational intensity or memory intensity. Majority of scheduling algorithms map computation intensive threads to bigger cores and some threads might monopolize the bigger cores. Hence, a case of proportional-share scheduling of threads is studied to determine performance improvement and energy savings. Runtime performance monitoring is used to give tickets

to threads. This ticket is used to schedule a thread on a big core based on lottery scheduling.

An asymmetric quad-core x86_64 chip which has individual core frequency scaling and four cores (one at 3.2GHz and three at 0.8Ghz) is used for validation of proposed techniques. The threads receive a dynamic number of tickets based on the energy efficiency ratio between scheduling the thread on a big core vs. on a small core. A lottery schedules uses this tickets to assign threads to cores at periodic intervals. The core power consumption for different workloads is taken into account for making energy-efficient scheduling decisions. The results show energy saving and performance improvements (ranging from 12% to 51%) when compared with biased and fair scheduling schemes used for heterogeneous systems when workloads taken into consideration are similar in nature.

However, due to resource unaware nature of policy, performance variations exist.

3.2. Performance Based Scheduler

The optimal use of heterogeneous multi-core depends on how well a scheduling policy matches workloads to the respective core type (big or small). Incorrect scheduling decisions can very well degrade performance and waste power as well as energy. Memory intensity along with ILP can be used to determine conditions based on which scheduling decisions can be made. Performance Impact Estimation (PIE) model can be used to take these parameters into a single metric and then map work load to a given core. Dynamic scheduling using PIE uses CPI stack as well as data about MLP (memory level parallelism) in order to efficiently make scheduling decisions. Additionally, shared LLCs which provide MLP data at low overheads (using lower sampling rates and PMU) assist in fine grained scheduling of tasks while accounting for time-varying conditions.

PIE is easily scalable to larger systems based with multiple cores and outperforms lottery based policy in terms of performance. For a wide variety of workload mixes PIE consistently achieves higher speed-up and can also be used to improve multi-threaded workload performance. PIE also takes fairness into consideration at certain levels; however tests to validate the same have not been conducted yet.

3.3. Comprehensive Scheduler

AMP are proposed as an alternative to SMP where the architecture has two types of cores. The fast cores are complex with high degree of ILP, higher frequency of operation prefacers and aggressive branch prediction. The slower cores have simple pipelines/design, smaller size and lower operating frequencies. Hence the system usually consists of lesser number of fast and higher numbers of small cores. When compared with an SMP system, AMP were found to deliver 60% more performance per watt.

In an SMP architecture, the resources will consume more power in case of parallel workloads for a complex system and incur high performance deficit for sequential workloads in a simple system. AMP provides a mix of both and hence caters itself as a useful alternative.

In order to make most of the resources provide by AMP system, the scheduling policies have to be carefully selected. CAMP targets reduction in energy cost and improvement in efficiency/ parallelism in AMP systems. A metric namely Utility Factor (UF) is used to evaluate the energy efficiency and TLP (thread level parallelism) of a particular application. Decisions of scheduling are then made using this metric. In case QoS is the goal, UF can be used as the complementary parameter to ensure efficiency while maintaining priority as per the QoS guarantee. Since, CAMP considers both efficiency and TLP; it performs better for a diverse range of workload when compared with previous AMP schedulers which target either efficiency or TLP improvement.

Based on the Utility factor of a thread, threads are placed in three bins i.e. Low, Medium and High. Initially all High utility class threads are mapped to fast cores. If threads in the high utility class are more than the number of fast cores then a round robin policy is adopted. Threads which fall under Low and Medium utility class are scheduled on slower cores. In case fast cores remain after mapping High utility classes, then medium class are scheduled on the fast cores and then low class.

Parallel applications (in the High class) with sequential phases are assigned to a special class so that they receive the needed boost using fast cores during the sequential execution period. Based on system thresholds, the utility thresholds are dynamically adjusted to have fine grained control on class assignment.

When compared with algorithms such as Parallelism-Aware (PA), which addresses TLP specialization, Speedup Factor-Driven (SFD), which addresses efficiency specialization, and round-robin (RR) for single threaded applications, CAMP and SF perform better in energy efficiency since PA is unaware of energy efficiency.

In case of a wide variety of workloads including combinations of single-threaded and multi-threaded applications, CAMP delivers performance gains since for sequential phases of all applications, CAMP initially runs them on fast cores and later degrades them as per classification. However, PA maps memory intensive applications to faster cores and parallel applications to slower cores from beginning and SFD maps parallel applications to faster cores failing to speedup initial sequential phase of the slower applications. Hence over a widely varying workload CAMP performs better compared to schemes concentrating at addressing only efficiency or TLP without any modifications.

4. COMPARISONS

With multi-cores becoming ubiquitous in computing, concurrent execution of many applications has become quite popular. This causes contention for shared resources. Since the on-chip LLC is at the periphery of resources on the multi-core processor, it is one of the last avenues to tackle majority of delays before the application issues a request to the slower main-memory.

In SMP systems CRUISE proves to gain maximum performance improvement and at the same time maintains energy efficiency. It does this by using both hardware and software support. Although hardware changes have been suggested, they are trivial and can be easily integrated into current systems. Other policies either tackle performance but are agnostic to lower level scheduling policies or try to gain maximum energy efficiency and cause performance to be less than optimal for workloads. HW/SW partitioning policy requires FPGAs and complex programming to convert high latency tasks into hardware executions. BWS targets work-stealing and is majorly aimed at commercial systems with higher workloads.

In AMP systems the comprehensive scheduler proves to be a major improvement over previous designs. It considers both energy efficiency and performance while accelerating application performance using asymmetric cores; hence it provides better results for a diverse range of workloads. Previous techniques only

solve one of the two problems. Hence their performance suffers when a mix of workloads with single and multi-threaded applications is provided.

5. REFERENCES

1. Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 2010. Addressing shared resource contention in multicore processors via scheduling. *In Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems* (ASPLOS XV). ACM, New York, NY, USA, 129-142. DOI=10.1145/1736020.1736036 http://doi.acm.org/10.1145/1736020
2. Andreas Merkel, Jan Stoess, and Frank Bellosa. 2010. Resource-conscious scheduling for energy efficiency on multicore processors. *In Proceedings of the 5th European conference on Computer systems* (EuroSys '10). ACM, New York, NY, USA, 153-166. DOI=10.1145/1755913.1755930 http://doi.acm.org/10.1145/1755913.1755930
3. Hassan A. Youness, Abdel-Moniem Wahdan, Mohammed Hassan, Ashraf Salem, Mohammed Moness, Keishi Sakanushi, Yoshinori Takeuchi, Masaharu Imai: Efficient partitioning technique on multiple cores based on optimal scheduling and mapping algorithm. *ISCAS 2010*: 3729-3732
4. Vinicius Petrucci , Orlando Loques , Daniel Mossé, Lucky scheduling for energy-efficient heterogeneous multi-core systems, *Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems*, p.7-7, October 07, 2012, Hollywood, CA
5. Aamer Jaleel, Hashem H. Najaf-abadi, Samantika Subramaniam, Simon C. Steely, and Joel Emer. 2012. 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). ACM, New York, NY, USA, 249-260. DOI=10.1145/2150976.2151003 http://doi.acm.org/10.1145/2150976.2151003
6. Xiaoning Ding, Kaibo Wang, Phillip B. Gibbons, and Xiaodong Zhang. 2012. BWS: balanced work stealing for time-sharing multicores. *In Proceedings of the 7th ACM european conference on Computer Systems* (EuroSys '12). ACM, New York, NY, USA, 365-378. DOI=10.1145/2168836.2168873 http://doi.acm.org/10.1145/2168836.2168873
7. Juan Carlos Saez, Manuel Prieto, Alexandra Fedorova, and Sergey Blagodurov. 2010. A comprehensive scheduler for asymmetric multicore systems. *In Proceedings of the 5th European conference on Computer systems* (EuroSys '10). ACM, New York, NY, USA, 139-152. DOI=10.1145/1755913.1755929 http://doi.acm.org/10.1145/1755913.1755929
8. Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. 2012. Scheduling heterogeneous multi-cores through Performance Impact Estimation (PIE*). In Proceedings of the 39th Annual International Symposium on Computer Architecture* (ISCA '12). IEEE Computer Society, Washington, DC, USA, 213-224
9. José A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. 2012. Bottleneck identification and scheduling in multithreaded applications. *In Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS XVII). ACM, New York, NY, USA, 223-234. DOI=10.1145/2150976.2151001 http://doi.acm.org/10.1145/2150976.2151001
10. Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh, and Onur Mutlu. 2012. Staged memory scheduling: achieving high performance and scalability in heterogeneous systems. *In Proceedings of the 39th Annual International Symposium on Computer Architecture* (ISCA '12). IEEE Computer Society, Washington, DC, USA, 416-427.
11. D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In HPCA '05: *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 340–351, 2005.
12. Y. Xie and G. Loh. Dynamic Classification of Program Memory Behaviors in CMPs. *In Proc. of CMP-MSI*, held in conjunction with ISCA-35, 2008.
13. R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro*, 28(3):54–66, 2008
14. Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and Approxima-tion 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, 2008
15. D. Tam, R. Azimi, andM. Stumm. Thread Clustering: Sharing-Aware Acheduling on SMP-CMP-SMT Multiprocessors. *In Proceedings of the 2nd ACM European Conference on Computer Systems* (EuroSys'07), 2007
16. J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. *ICS-21*, 2007
17. A. Jaleel, E. Borch, M. Bhandaru, S. Steely, and J. Emer. Achieving Non-Inclusive Cache Performance With Inclusive Caches -- Temporal Locality Aware (TLA) Cache Management Policies, *In MICRO*, 2010
18. M. K. Qureshi and Y. Patt. Utility Based Cache Partitioning: A Low Overhead High-Performance Runtime Mechanism to Partition Shared Caches. *In MICRO-39*, 2006