Cache Friendly and Capacity Conscious Scheduling in Multi-core Systems

Sheela Kathavate
Department of Computer Science and Engineering
Sir M. Visvesvaraya Institute of Technology
Bangalore

N. K. Srinath
Department of Computer Science and Engineering
R. V. College of Engineering
Bangalore

ABSTRACT
Current generation high performance multi-core processors have large shared cache memories. This shared cache memory is accessible by multiple cores. Concurrently running threads under each core do not always demand the entire capacity of the shared cache. Threads running on different cores accessing shared cache concurrently may result in higher cache miss rate and significant performance degradation due to inter-thread cache conflicts and lack of cache space. The cache capacity is the quantity of physical cache memory available with the processor. To achieve certain higher degree of processing performance on multi-core processors, efficient shared cache memory usage plays the defining role. The overall processor performance gets more sensitive to the problem of shortage of cache capacity, as threads sharing the cache compete for their requirement of the cache sizes. In this paper, a cache friendly and capacity conscious thread scheduling strategy is proposed for multi-core processors with multiple shared caches. The proposed scheduling policy ensures that the shared cache is optimally used by the competing threads which minimizes inter-thread resource conflict and hence reduces performance degradation. According to the experimental results the proposed policy reduces shared cache contention significantly thereby improving the overall performance among threads by up to 5%.

Keywords
CMP, Cache Capacity, Thread Scheduling, Shared Cache.

1. INTRODUCTION
The current trend in processor chip manufacturing technology is experiencing the limitation of thermal diffusion technology, physical characteristic and semiconductor processes. As a result, the speed of the processor is expected not to double as per Moore's law. To overcome this limitation in processor scaling capability, Chip Multi-Processor (CMP) has become the trend and is prevalent in modern computer systems. Multiple threads can run concurrently on a CMP consisting of multiple cores. Each core is embedded with its private cache called level-1 (L1 Cache) to share among the threads of individual core. The last level cache (LLC Cache) is shared among multiple cores to enhance cache resource utilization.

Modern multi-core systems are designed to allow clusters of cores to share various hardware structures, such as a LLC memory controllers, and interconnects, as well as prefetching hardware. Figure 1 shows a multi-core system with each core having a private L1 cache and a shared last level L2 cache between two cores. Concurrently running threads, or co-runners often share a single second level (L2) cache, and cache allocation is controlled by the underlying hardware [1]. Cache sharing depends solely on the cache needs of the co-runners, and unfair cache sharing occurs often. When the shared cache is being accessed by applications running on different cores, cache miss could occur resulting in the degradation of system performance. Hence, addressing shared cache contention issue in CMPs becomes an issue to be addressed.

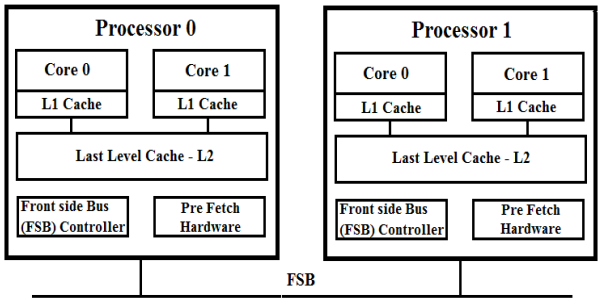

Figure 1: Block Diagram of 4 Core CMP architecture

Concurrently executing threads spawned from different applications on multi-cores cause performance degradation issues due to the contention of shared resources among threads. As a result of this, the time taken to execute applications on a multi-core processor may consume longer time than that of a uni-core processor, even though the former can simultaneously execute multiple threads. Hence, avoiding inter-thread resource conflicts becomes a major task for a multi-core processor.

The rest of this paper is structured as follows. In Section 2, related work is discussed. Section 3 describes cache conscious scheduling algorithm, Section 4 explains experimental methodology. Section 5 shows Results and analysis and conclusion in Section 6.

2. RELATED WORK
To overcome the shared cache contention, many researchers have proposed scheduling policies considering the inter-task interference. J. Mars et al. [2] proposed a framework (CiPE – Cross-core interference Profiling Environment) which is composed of a lightweight runtime environment on which a host application runs on one core. With this, a contention synthesis engine executes on a neighboring core. CiPE manipulates the co-running contention synthesis engine, while monitoring and analyzing the resulting dynamic impact on the host application. Here, a specially designed benchmark was used to be co-scheduled with another task. The throughput of the task was used to assign scores to itself. In this methodology, the cache capacity is not taken into consideration. Xie et al. [3] proposed classification of an application based on its anti-interference abilities. Anti-interference is defined as the loss in performance degradation when one application competes with the other for shared cache. This method classifies the tasks analogous to animal personalities, such as sheep, devils, rabbits and turtles, based on some metrics related to the cache. In order to reduce the

cache contention, the authors proposed co-scheduling of a task having better anti-interference ability with a less anti-interference ability task. This method of scheduling may not be accurate as it is difficult to co-schedule task which have similar anti-interference abilities.

G. E. Suh et al. [4] described a cache partitioning algorithm centered on the concept of low overhead control scheme and marginal gain. However, the drawback in this algorithm is the accountability of fairness in sharing of cache among threads. M. K. Qureshi et al. [5] and S. Kim et al. [6] have proposed mechanisms to reduce inter-thread cache conflicts on last level cache using dynamic cache partitioning concept. They defined the fairness among threads as parity of performance degradation. As per their algorithm, the allocation of partitioned cache capacities is decided based on the minimum difference in miss rates among running threads. Such mechanisms have shown certain amount of performance improvement. However, as dynamic cache partitioning does not consider the cache capacity of each thread, performance severely degrades.

Based on the dynamic behavior of the cache, some research has been done in the field of thread scheduling on multi-core systems. Jia *et al*. [7] used L2 MRCs (Miss Rate Curve) to classify applications based on their cache behavior dynamically. These methods need special hardware, like LRU counters and ATD. James H. Anderson et al. [8] proposed a cache aware Pfare-based scheduling scheme for real time task on multi-core platforms, but they have only considered static and independent task. Shekofteh *et al*. [9] proposed a risk function and approximated the probable cache contention of a co-schedule and the schedule with the least risk was chosen.

When threads execute concurrently, conflict misses occur among threads whenever one thread replaces the data of the other in cache. J. Kihm et al. [10] refer to such conflict as *inter-thread kickouts* resulting in increased execution time of a thread. To avoid such inter-thread kickouts, dynamic cache partitioning has been proposed. Dynamic cache partitioning divides the shared cache memory into multiple parts of different sizes and each thread receives its share at run-time. Carol-Jean Wu et al. [11] considers operating system priority levels as part of capacity management. The proposed methodology uses time keeping techniques that track an account of the time between two cache accesses. This technique needs to have a knowledge of the priority of each application. Sharanyan Srikanthan et al. [12] have proposed SAM, a Sharing-Aware Mapper that uses the aggregated coherence and bandwidth event counts which is used to separate traffic caused by data sharing due to memory accesses. In this methodology, cache sharing is not considered. Baptiste Lepers et al. [13] discuss about optimistic multi-core schedulers wherein the load balancing for the cores is taken into consideration.

Furthermore, if cache sharing threads request large size cache capacities, it could result in degradation of performance under dynamic cache partitioning. For example, consider that two threads that are running on a shared cache. When the combined cache capacities requested by these two threads surpasses the capacity of the shared cache, the dynamic cache partitioning fails in meeting their demands. This results in severe performance degradation among concurrently executing threads. Even though dynamic cache partitioning is effective in avoiding the problem of inter thread kickouts, it fails to address the capacity contention issue.

In this paper, a cache conscious scheduling algorithm is proposed. This proposal considers the cache requirement of the individual threads and schedules them dynamically to avoid cache contention and eventually improves the overall system performance.

3. CACHE Conscious THREAD SCHEDULING POLICY

3.1 Motivation

When threads are not classified according to their shared cache requirements, the scheduling of such threads will result in cache resource depletion or create unused cache resource. In Figure 2, Thread 0 and Thread 1 have been assigned to LLC 0 which results in cache depletion while Thread 2 and Thread 3 assignment to LLC 1 results in unused cache resource. This would have done better if the scheduler allocated Thread 0 and Thread 3 to share one cache (LLC 0) while allocating Thread 1 and Thread 2 to share the other cache (LLC 1) to optimize the available cache as shown in Figure 3.

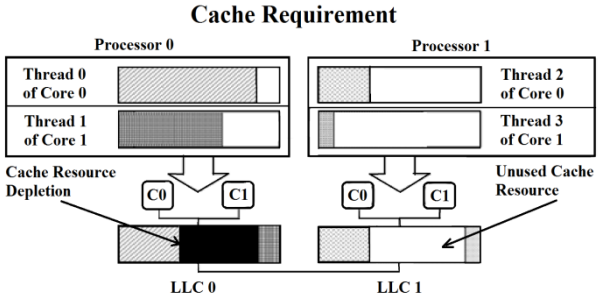

Figure 2: Scheduling with the cache capacity shortage and cache resource being unused

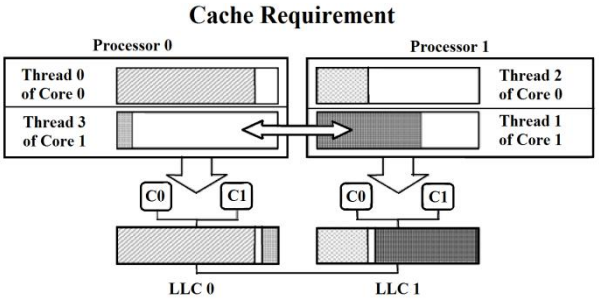

Figure 3: Scheduling by cache conscious thread scheduling policy

A classification and scheduling algorithm which takes into consideration the cache requirements of each thread and group them before distributing to the cores is proposed. The proposed scheduling policy combines thread of higher cache requirement with that of thread which needs lower cache requirement to run on shared cache.

3.2 Profiling for Cache Requirement of a thread

It is important to analyze and evaluate the request for cache capacity of every thread during profiling phase. The underlying methodology uses a profiling scheme that controls the cache partitioning mechanism as applied to every shared cache. In this mechanism the stack distance profiling is used to decide the number of cache ways to be allocated for each thread.

Figure 4 illustrates the low locality data access as per Chandra et al. [14]. In this graphical representation C_x represents the number of accesses to the blocks at the x^{th} LRU position. Low locality means accesses are spread out all over the positions. This infers that the available cache capacity is lesser than the capacity needed by the thread. On the contrary, a thread having high locality means the requested capacity by a thread is lesser than the cache capacity. This high locality occurs frequently at the adjacent positions as shown in Figure 5. To mathematically analyze and to quantify this locality difference, H. Kobayashi et al. [15] defined an assessment matrix '*D'* as C_1/C_N. The scheme combines the assessment matrix *D* and predefined thresholds, t_1 and t_2 for the purpose of cache resizing. When the assessment matrix *D* becomes greater than threshold t_2, the cache partitioning scheme assigns one additional way for the active thread. On a differing note, when assessment matrix *D* becomes smaller than threshold t_1, the scheme removes or de-allocates a way to the active thread.

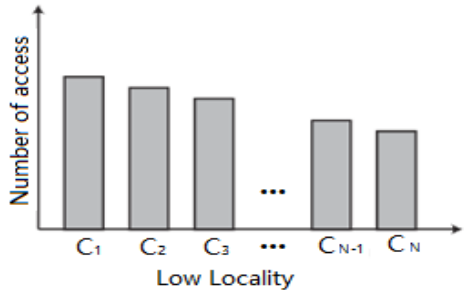

Figure 4: Low Locality Stack Distance Profiling

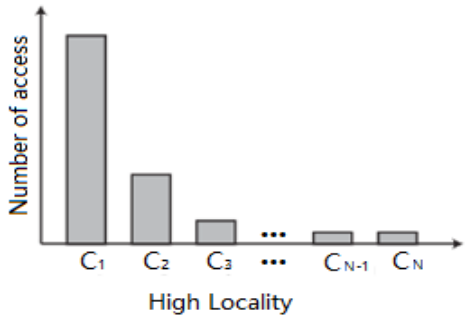

Figure 5: High Locality Stack Distance Profiling

The cache Capacity Requirement of a thread is derived from:

$$CR = \frac{\sum_{t=t0}^{t1} W(t)}{t1 - t0} \quad \text{-- (1)}$$

Where,

$W(t)$ = (ways allocated at time t)

t_0 = (the starting time of profiling)

t_1 = (the ending time of profiling)

3.3 Thread Scheduling Algorithm

In Algorithm 1, m threads running concurrently on an n-core processor with n/2 shared caches. Each L2 cache is shared by two cores and every thread is associated with a cache requirement. Let T be an array of m threads. The threads in this array T are sorted by their cache requirement in descending order. The top m/2 threads are scheduled on to the first core (C0) of the corresponding shared cache Si (lines 2, 3 of the algorithm). The scheduled threads are deleted from the list T. Same steps are taken for the remaining threads in T but making sure that the threads are scheduled in the reverse order of the shared caches (lines 6 and 7). This makes sure that the thread with the highest cache requirement is scheduled with the lowest cache requirement thread to use the shared cache.

Algorithm 1: Thread Scheduler

1. Sort **m** threads of T in descending order of their Cache Requirement (CR)
2. **for** i = 1 to n/2 **do**
3. assign the first thread in T to C_0 of S_i;
4. m = m – 1; //remove the assigned thread**end for**
5. **for** i = n/2 to 1 **do** //assign remaining threads to C_1 core of S_i
6. assign the first *thread in T* to C_1 core of S_i;
7. m = m - 1; //remove the assigned thread
8. **end for**

4. EXPERIMENTAL METHODOLOGY

The effectiveness of the proposed cache conscious scheduling algorithm has been evaluated using the Gem5 simulator [16]. A four core CMP with 2 shared LLCs is used as the baseline system and single threaded cores with L1 private cache. The L1 instruction cache and Data cache are 4-way 32 KB each, while the last level cache is a unified 32 way 4 MB cache. For the evaluation, six benchmarks from SPEC CPU2006 suite based on the characteristics of cache accesses are used. The description of these benchmarks are given in Table 1. Table 2 also lists the same six SPEC CPU2006 benchmark programs and their Misses Per Kilo Instruction (MPKI) in the L1 cache and LLC when run in isolation. MPKI values give a measure of the cache utility for an application. In the selection, all the benchmarks are categorized into three types namely, high-utility, saturating-utility and low-utility based on their MPKI values. Benchmarks *mcf* and *libquantum* are categorized into high-utility benchmarks. These applications have a working set size that is greater than the available LLC and their performance increases gradually as cache size increases. *Astar* and *bzip* are categorized into saturating-utility. Their performance saturates with a smaller cache size and will not improve when the cache size is increased. *Perlbench* and *dea* are categorized into low-utility. They benefit very less from the last level cache as their working set always fits into the lower level caches.

The Cache Requirement (CR) analysis of the six CPU SPEC 2006 benchmarks calculated using the stack distance profiling method as explained in the previous section matched closely with that of MPKI analysis as in Table 2. This justifies the usage of cache requirement analysis using stack distance profiling in the proposed scheduling algorithm.

Table 1: SPEC CPU2006 Benchmarks and their characteristics

Benchmark	Description
bzip	Data Compression
astar	Path Finding
dea	Solution of partial differential equation using adaptive finite element
perlbench	Perl programming language
libquantum	Physics / Quantum computing
mcf	Combinatorial optimization /single depot vehicle scheduling

Table 2: MPKI of Representative SPEC CPU 2006 benchmark applications

Benchmarks →	**astar**	**bzip**	**dea**	**perlbench**	**libquantum**	**mcf**
L1 MPKI (64 KB)	29.29	19.48	0.95	0.42	38.83	21.51
LLC MPKI (4MB)	2.02	1.05	0.05	0.60	34.28	18.72
Classification	SAT	SAT	LOW	LOW	HIGHH	HIGH
Cache Required	**17**	**10**	**8**	**6**	**31**	**32**

High=High Utility, **Sat**=Saturating Utility, **Low**=Low Utility

To evaluate the proposed scheduling algorithm, six benchmark combinations of four threads each has been formulated. The best possible combination matrix of six workloads is selected out of the possible 15 combinations and are listed in Table 3. These workloads are simulated using the proposed scheduling algorithm. A worst application schedule which assigns two high cache requirement applications to the shared cache was also simulated for comparison purpose.

In order to measure the performance of concurrently executing applications, the 'Harmonic Mean Fairness' metric has been used. The harmonic mean fairness metric is the harmonic mean of normalized IPCs which balances both performance and fairness [17]. This metric is obtained as follows:

$$\text{Harmonic Mean Fairness} = \frac{N}{\sum_{i=0}^{N-1}\left(\frac{Solo\text{-}IPCi}{IPCi}\right)} \quad \text{-- (2)}$$

Where N represents the number of threads, IPCi is i^{th} application's IPC when it concurrently runs with other applications. *Solo-IPCi* is i^{th} application's IPC running in isolation.

Table 3: Benchmark combinations by thread characteristics

Work loads	Applications			
WL1	libquantu	mcf	dea	perlbench
WL2	libquantu	mcf	astar	dea
WL3	libquantu	mcf	astar	bzip
WL4	libquantu	astar	bzip	perlbench
WL5	libquantu	bzip	dea	perlbench
WL6	astar	bzip	dea	perlbench

5. RESULTS AND ANALYSIS

Figure 6 shows the performance of different workloads over Default Application Scheduling (DAS). The proposed policy shows an overall performance improvement of 5%. This implies a proper combination of workloads that can run on a shared cache is the key factor of system performance improvement. As per this policy, WL1 achieves the highest performance of 8% compared with the default application scheduling. This consists threads of combination with two lowest and two highest cache requirement. A severe performance degradation can be observed in the worst case scenario in which two threads having the highest cache requirements are co-scheduled in the same shared cache and two other threads having the lowest cache requirements are co-scheduled in the other shared cache. Here, the shared cache either has cache shortage or waste of its cache capacity resource.

The proposed scheduling policy effectively prevents this unfair situation by allocating different shared caches for the two high cache requirement threads. With respect to the WL6 combination, the proposed scheduling policy doesn't perform well, as the threads in this group have saturating utility and low utility cache requirement and a proper cache space can be allocated to them by the default scheduler. The results show that the cache conscious scheduling policy performs better than the default case policy in other workload cases. If the cache capacity has not been considered, dynamic cache partitioning would degrade the thread performance significantly. Therefore, the proposed scheduling policy has shown to be effective in preventing this problem.

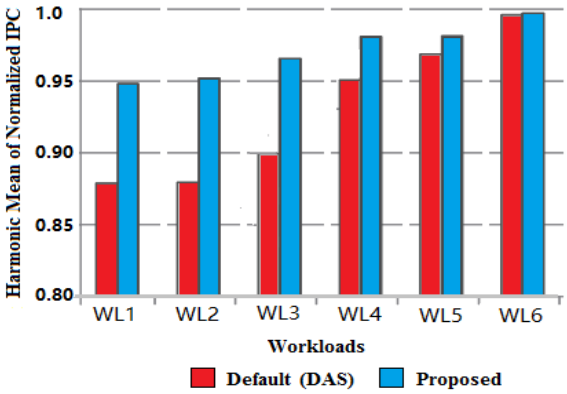

Figure 6: Performance Comparison on Proposed Policy to Default Application Schedule (DAS)

6. CONCLUSION

In this paper, a cache conscious friendly scheduling policy is proposed. The policy classifies the threads based on their cache requirements and schedules them intelligently to the shared last level caches. This ensures the threads demanding higher cache ways are distributed uniformly among the different shared LLCs along with threads that require less cache ways, so that the available cache space is optimally allocated. Accordingly, the shared cache contention is reduced and the performance improves. The contiguous cache memory allocation for the application threads in the shared last level cache is ensured by this proposed scheduler. This makes it Cache Friendly in nature to the application whether it's classified in the range as a high utility or as a low utility thread.

Experimental results show a performance improvement of 8% for a particular workload and an average of 5% for the proposed scheduling algorithm compared to the default application scheduling.

Future work involves developing a dynamic profiling and scheduling algorithm which can take care of the requirements of the applications in real time.

7. REFERENCES

[1] P. Kongetira, et al, 2005 "A 32-Way Multithreaded SPARC Processor", IEEE Micro, vol. 25 Mar, (2005).

[2] J. Mars, L. Tang, and M. L. Soffa, 2011 "Directly characterizing cross core interference through contention synthesis", In Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers, Pages 167–176.

[3] Y. Xie and G. H. Loh, 2008 "Dynamic classification of program memory behaviors in CMPs", In *Proceedings of CMP-MSI*.

[4] Ed Suh, Larry Rudolph, Srini Devadas, 2001 "Dynamic cache partitioning for simultaneous multithreading systems", in the proceedings of 13th IASTED International Conf. on Parallel and Distributed Computing and Systems, 116–127.

[5] M. K. Qureshi and Y. N. Patt, 2006 "Utility-based cache partitioning: A low-overhead, high-performance, run-time mechanism to partition shared caches", in the proceedings of 39th Annual IEEE/ACM International Symposium on Microarchitecture, 423–432.

[6] S. Kim, D. Chandra, and Y. Solihin, 2004 "Fair cache sharing and partitioning in a chip multiprocessor architecture", in the proceedings of 13th International Conference on Parallel Architecture and Compilation Techniques, 111–122,

[7] X. Jia, J. Jiang, T. Zhao, S. Qi, and M. Zhang, 2010 "Towards online application cache behaviors identification in CMPs", in the *Proceedings of the 12th IEEE International Conference on High Performance Computing and Communications*, pages 1 – 8.

[8] James H. Anderson, J M Calendrino, and U C Devi, 2006 "Real time scheduling on multi core platforms", in the proceedings of the 12th IEEE *Real Time and Embedded Technology and Applications Symposium (RTAS '06),* San Jose California, USA, April 4-7.

[9] S. Shekofteh, H. Deldari, and M. B. Khalkhali, 2010 "Reducing cache contention in a multi-core processor via a scheduler", *International Conference on Advanced Computer Theory and Engineering*.

[10] J. Kihm, A. Settle, A. Janiszewski, and D. Connors, 2005 "Understanding the impact of inter-thread cache interference on ILP in modern SMT processors", *The Journal of Instruction-Level Parallelism, 7*.

[11] Carol-Jean Wu, Margaret Martonosi, 2011 "Adaptive Timekeeping Replacement: Fine Grained Capacity Management for Shared CMP Caches", ACM Transactions on Architecture and Code Optimization (TACO), Vol. 8, Issue 1, Article 3.

[12] Sharanyan Srikanthan, Sandhya Dwarkadas, and Kai Shen 2015, "Data sharing or resource contention: toward performance transparency on multicore systems", in the Proceedings of the 2015 USENIX Annual Technical Conference (USENIC ATC '15)", July 8–10, 2015, Santa Clara, CA, USA, pp. 529-540.

[13] Baptiste Lepers, Willy Zwaenepoel, Jean-Pierre Lozi, Nicolas Palix, Redha Gouicem, 2017 "Towards Proving Optimistic Multicore Schedulers", HotOS 2017 - 16th Workshop on Hot Topics in Operating Systems, ACM SIGOPS, Whistler, British Columbia, Canada.

[14] D. Chandra, F. Guo, S. Kim, and Y. Solihin, 2005 "Predicting inter-thread cache contention on a chip multi-processor architecture", In the proceedings of 11th International Symposium on High Performance Computer Architecture, pp.340–351.

[15] H. Kobayashi, I. Kotera, and H. Takizawa 2005 "Locality analysis to control dynamically way-adaptable caches," ACM SIGARCH Computer Architecture News, vol.33, no.3, pp.25–32.

[16] Nathan Binkert, Bradford Beckman, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek. R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill and David A. Wood. 2011 "The gem5 simulator", ACM SIGARCH Computer Architecture News, Vol. 39, No. 2 pp. 1-7.

[17] K. Luo, J. Gummaraju, and M. Franklin, 2001 "Balancing throughput and fairness in SMT processors", In ISPASS, pages 164–171.