http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 12 (7): 1366-1372, 2013
ISSN 1812-5638 / DOI: 10.3923/itj.2013.1366.1372
© 2013 Asian Network for Scientific Information

Critical Thread Guided Fine-grained Adaptive Capacity Management for Shared CMP Caches

Xin Xu and Manman Peng
Xin Xu School of Information Science and Engineering,
Hunan University, No. 252, Lushan South Road, Changsha, 410082, China

Abstract: With the shift towards Chip Multiprocessors (CMPs), load imbalance between the different CPU due to causes not controlled by the application developer, resulting in significant performance degradation and waste of CPU time. Although there are many techniques to address load imbalance at run-time, as it happens, these techniques may not be particularly effective when the cause of the imbalance is due to the performance sensitivity of the parallel threads when accessing a shared cache. To this end, we present a novel run-time mechanism, using criticality prediction to guide cache space allocate, with minimal hardware, that automatically tries to balance parallel applications using dynamic cache allocation. The mechanism detects which thread is critical and reduces imbalance by assigning more cache space to the slowest threads. This experiment on a detailed microprocessor simulator with the Computer Vision and Data Mining applications reveal that this scheme can improve performance from 1-6%. Fine-grained temporal control is particularly important for parallel applications which are expected to be increasingly prevalent in years to come.

Key words: Load balancing, critical thread, cache allocation, OpenMP

INTRODUCTION

While Chip Multiprocessors (CMPs) already dominate computer systems, key research issues remain in exposing managing the parallelism required to fully exploit them. In order to obtain good performance from parallel applications it is essential to guarantee that, during execution, the amount of time a processor is waiting for other processors' results is kept at a minimum. Programmers spend a lot of time to achieve the load balance between different threads.

According to Boneti *et al*. (2008a), during the application's execution, intrinsic and extrinsic load imbalance is the root cause of thread's load imbalance. Intrinsic load imbalance is caused by characteristics, such as the input data, different sub-domain, date exchanging etc which are intrinsic to the application. For example, a thread has a small input set to work on while another has a large amount of data to process; Threads exchange data among themselves require different time. It is highly difficult for the application programmer to balance the application a priori to deal with all possible input data sets. Even though the application's input set are balanced, the execution of the parallel application could still be imbalanced (Boneti *et al*., 2008b). External factors which may slow down some threads but others, lead to extrinsic load imbalance. A major source of load imbalance could be the operating system noise when performing services such as handling interrupts etc. Also, extrinsic load imbalance may be caused by threads contention for processor's shared resources, such as shared last level cache. Obviously, there is nothing that the application programmer could do a priori to prevent extrinsic load imbalance.

In order to balance the work load among different threads, one feasible solution is to move some workload from threads whose execution more slowly to fast-running threads; another is to allocate more resources to slow-running threads dynamically (Su *et al*., 2012). Such resources are mainly the CPU time slices, with more CPU time slices assigned to slow-running threads (Bhattacharjee and Martonosi, 2009). Other authors make use of hardware thread priorities to change instruction's decode rate of each thread running on an SMT architecture (Cai *et al*., 2008). Processor resource allocation controlled by software that allows balancing parallel applications, but decode priority almost doesn't work on memory-bound applications (Boneti *et al*., 2008a) and many CMP processors have only one thread per core.

Adding more CPU time slices to slow-running threads or adjusting the workload of each thread may seem a natural solution to the problem. However,

Corresponding Author: Manman Peng, School of Information Science and Engineering, Hunan University, No. 252, Lushan South Road, Changsha, 410082, China

uniformly resorting to such solutions may forfeit the possible benefits stemming from other options which may sometimes be potentially closer to the root of the problem. Most CMP system is characterized by shared last level cache, the sharing improve the performance among cooperative threads, but also results in cache contention for parallel applications. And the competition on the shared cache caused the speed of different thread reach the barrier early or late. The slowest thread reach the barrier last, other threads have to wait for it, so, the slowest thread correspond to the critical thread.

In this study, we present a novel mechanism, critical thread guided fine-grained adaptive capacity management, as we will demonstrate later in this study, find out the critical thread and allocation more cache space to it, finally, achieving load balance between different threads and speedup parallel application, show that our mechanism can improve performance for various workloads, from 1-6%.

IDENTIFICATION OF CRITICAL THREADS

We aim at detecting dynamically the workload imbalance of parallel applications. Figure 1 demonstrates that even very regular parallel programs may exhibit workload imbalance during execution. Figure 1a shows the pattern of OpenMP parallel application. The code is already written in such a way that the input data set is partitioned to achieve workload balance. Assuming there are n core in CMP system, total workload will evenly distributed to n chunks automatically by OpenMP compiler, a chunk processed by one thread. However, workload imbalance still exists. Reasons for workload imbalance could be intrinsic or extrinsic factor.

We propose to identify the critical thread dynamically during program execution. According to (Bhattacharjee and Martonosi, 2009) which statistics last level cache miss ratio to find the critical thread, but this method is not precise measurements, sometimes the prediction is wrong, will lead to much performance loss. Assuming that the parallel workload follows some kind of an iterative pattern, where there is a repetition of a sequence consisting of a computation phase followed by a barrier synchronization, inserting check point at the back edge of a parallel loop is a natural choice, because the back edge of a loop is visited many times by all threads at runtime. Then the behavior of the application in the previous iterations can be used to make decisions for the next iteration.

To find the slowest thread, firstly, have to insert checkpoints. One candidate for a checking point is the place in a parallel region that is visited by all threads many

```
(a)
#pragma omp parallel for
    for (j = 0; j<mergedworknum; j++) {
        int thread = omp_get_thread_num();
        // a lot codes
        ......
        ......
    }   // end for
```

```
(b)
#pragma omp parallel for
    for (j = 0; j<mergedworknum; j++) {
        int thread = omp_get_thread_num();
        // a lot codes
        ......
        __asm ("count_op_inst")
    }   // end for
```

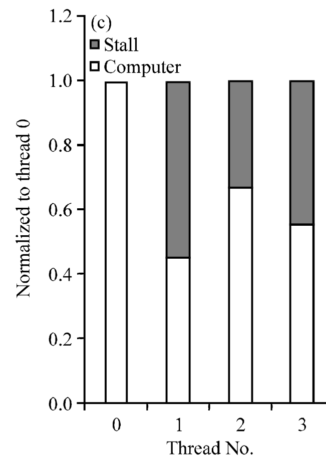

Fig. 1(a-c): (a) A parallelized loop from Freqmine (b) Insertion of a pseudo instruction as checking point (c) Profiling from one freqmine parallel region, CPU busy cycles normalized to thread 0

times during parallel execution. For example, in Fig. 1b, it is easy to see that the last statement of the outermost loop satisfies our criteria, then, check point inserted to the last statement of parallelized loop. While we count the number of checkpoints, the slowest thread is the one with the smallest counter and the speed of a thread can be estimated as the difference of its counter and the counter of the slowest.

The insertion of a checking point can be done by the hardware, the compiler or the programmer. A hardware approach, although it is completely transparent, it requires extra hardware structure to detect a suitable checking point among repeated instructions in a parallel execution. In this work, we add a pseudo instruction into simulator which could decode the pseudo instruction we insert into

the source code. The application is rewritten like Fig. 1b, the user inserts the check point and it identified by simulator which translates the new pseudo instruction that, once decoded, increments the private counter of the thread.

To take use of the checking point referred above, we need an array to record the number of iteration with different threads. This array has as many entries as number of cores in the processor. Each entry contains a 32-bit counter that keeps track of the number of times each core has reached the given instructions. When a core decodes a checking point, the counter corresponding to its assigned thread in the array is incremented by 1. Every n×10 times decoding of the checking point instruction, the processor stops fetching instructions and to execute the cache space allocate algorithm.

In the OpenMP programming model, if parallel codes are extraordinary irregular, dynamic scheduling can be used. Our critical thread identification is not suitable for this scenario. However, dynamic scheduling has large runtime overheads, static scheduling is recommended as the first scheduling option, especially when the number of threads is increased. The decision whether to use static or dynamic scheduling in a parallel is out of the scope of this study.

DYNAMIC LOAD BALANCING THROUGH CACHE ALLOCATION

In a fork-join parallel execution model such as OpenMP, a parallel loop usually has a barrier at the joint point of the loop that synchronizes all threads. In the best case, all threads reach this barrier at the same time. However, in a normal situation, some threads reach the barrier earlier than others and spend a lot of time waiting for slower ones, like Fig. 2a, three non-critical thread have to wait the critical thread. Ideally, assigning more cache resources to the slowest thread would reduce its execution time without excessively degrading the other threads' performance. Figure 2b shows the execution of the application after a new assignment of the cache resources has been applied to the critical thread, it is assumed that T'<T. So, it reduces the overall execution time of the application. In the case of multiple threads running on CMP systems, multiple requests are made to the shared cache simultaneously, however, current systems cannot explicitly assign shared cache resources effectively based on thread's speed. Our method is that, using mechanism described in section 2 to finds out the critical thread then activates a balancing algorithm later.

As we know, like LRU cache eviction policy, cache lines are associated with a counter which is decremented periodically over time. Each time a cache line is referenced, its counter is reset back to a specified interval, T which is the number of idle cycles this cache line is allowed to remain in the cache. When a new request arrives to a set that is full, LRU eviction policy chooses the victim line to evict from the cache set. The LRU eviction policy tends to give more cache space to threads visiting more frequently the shared last level cache. Moreover, when using LRU as eviction policy the operating system cannot control how threads share a cache effectively.

We suggest to evicting the data of fastest thread more aggressively than that of slowest thread. This approach can be used to control cache space occupancy of multiple threads, to achieve the load balance between different threads. So, we use different decay intervals for each thread. The decay counters of cache lines associated with critical thread hold a longer decay interval, so over time more cache space are allocated to these threads. Consequently, slower thread's data tend to stay in the shared cache for more long-term. For decay counter implementation, this scheme primarily relies on a single global cycle counter. The global counter's high or middle-order bits could be used to control the per cache line in the LLC, as discussed by Hu *et al*. (2002). Only a

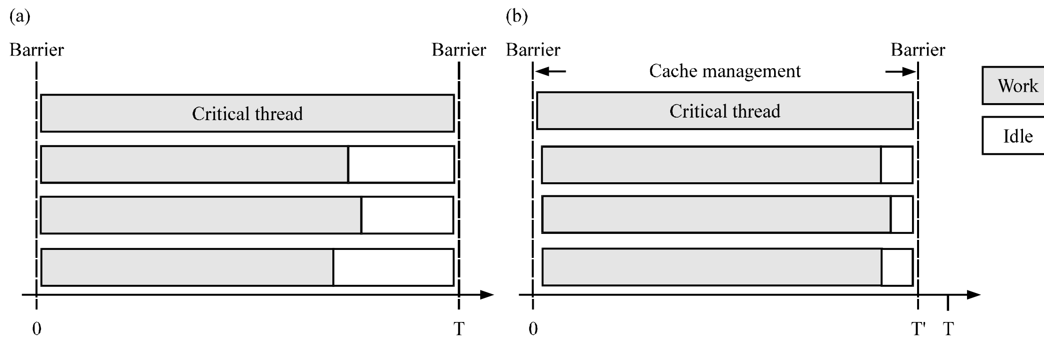

Fig. 2: Example of a parallel application with 4 threads running in the same CMP

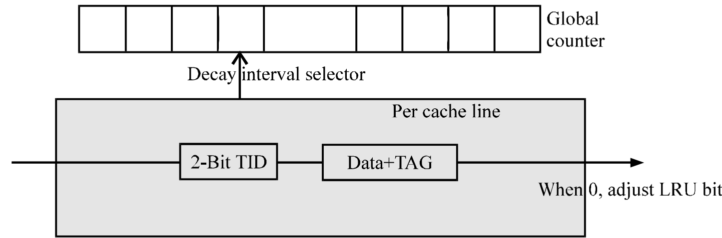

Fig. 3: Hardware implementation

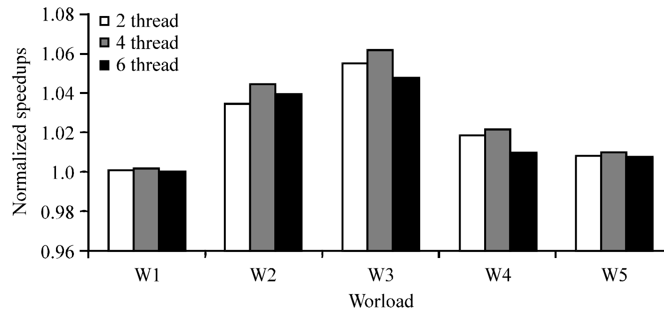

Fig. 4: Performance results for critical thread guided fine-grained shared CMP caches capacity management

small number of coarse-grained decay bits, e.g., 2 bits in our scheme, are stored per cache entry along with a subset/hash of TID.

Figure 3 depicts hardware implementation. The hardware interprets thread critical with an N entry lookup table, where N is the number of threads. This small hardware lookup table maps TIDs to decay intervals for all active threads. Then, at every sampling interval, compares the performance of critical thread and the non-critical thread. It also calculates the next decay intervals based on the comparison result and α values. Since we only allow α factor to be multiples of 2, the calculation for new decay intervals is a simple shift and, as a result, this calculation is not a performance bottleneck. Also, Decay counter implementation with 2-bit speed counter per cache line incurs less than 1% overhead for 64 byte cache lines (Fig. 4).

Table 1 provides the pseudo code decay interval calculation. There are two priority levels in our framework: critical and non-critical. With the two priority levels, there are three parameters. T_i refer to the iteration counter described Section 2, So, $T_{critical}$ stands for iteration number of slowest thread. Cache lines associated with critical threads never change, whereas, those associated with non-critical adjust. When non-critical thread run fast than critical thread over the threshold value δ, Decay interval of non-critical thread divide α, otherwise the decay interval of different threads in the same which evicting the victim cache line like LRU. Decay interval adjustments

Table 1: Pseudo code of decay interval calculation

```
if (Core P decode pseudo instruction "count_op_inst" )   {
   Update the critical counter of Core P
   if (critical counter of Core 0 mod 10 == 0)   {
        L ← list of threads sorted by critical counter number
        while (number of threads in L > = 2 ) do {
            if (Ti-Tcritical >= δ)
                DecayIntervali = DecayIntervali / α
            else
                DecayIntervali = DecayIntervalcritical
            remove the slowest and fastest threads from L
        } //end while
   } // end if
} // end if
/* Since we only allow factors α to be multiples of 2, the calculation
for new decay intervals is a simple shift with global counter */
```

continue dynamically until load balancing between different threads reach their pressed lower bounds. In general, the critical thread is allocated more shared cache space and can recover from the performance loss caused by the interference of others. After performing a sensitivity study, we concluded that good performance is obtained when initial Decay interval Cycle equal 65536, $\alpha = 2$ and $\delta = 3$.

EXPERIMENTAL ANALYSIS

In this study, we use gem5, a full-system multi-core processor Simulator, providing a flexible, modular simulation system that is capable of evaluating a broad rangeof systems and is widely available to all researchers

Table 2: CMP system configuration used

CPU	L1	L2	L3
Intel Atom D525, 2-core 4-thread,1.8GHz	32KB (I) 24KB (D), 4-way, 64B cache line, private	512KB, 16-way, 64B cache lines, shared	NONE
AMD Phenom II X6 1065T 6-core, 2.9GHz	64KB (I) 64KB (D), 2-way, 64B cache line, private	512KB, 16-way, 64B cache lines, private	6MB, 64-way shared

Table 3: Parallel application workload

Workload	Program	Application Domain	Parallelism	Working set
W1	Blackscholes	Financial analysis	Data-parallel	4,096 options
W2	Freqmine (Scan)	Data mining	Data-parallel	250,000 click
W3	Freqmine (Fpgrowth)	Data mining	Data-parallel	250,000 click
W4	bodytrace (CreateEdgeMap)	Computer vision	Data-parallel	1,000 particles
W5	bodytrace (CalcWeights)	Computer vision	Data-parallel	1,000 particles

which allows us to model CMP architectures. We simulate 4 and 6 core CMP systems (Table 2) based on the Alpha architecture running Linux 2.6 operating system.

Finally, we use PARSEC (Bienia *et al*., 2008) as the benchmark suite which is designed for CMP research. Bienia *et al*. (2008) have shown that the suite covers a wide range of working set sizes and a variety of locality patterns, data sharing, synchronization and off-chip traffic, making it an attractive choice over some old parallel benchmark suites such as SPLASH-2. We check all program accord with the characteristics of section 2 descripted, list them in Table 3 and run the benchmark in gem5 with 2 and 4 threads simulating Intel Atom cache hierarchy and 6 threads AMD Phenom cache hierarchy.

Blackscholes, is a financial application. It calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation. We tested two versions of BlackScholes, one is original and another is data race in Blackscholes. However, the program running times have no considerable changes. Through reading the program, we confirm that the program is a compute intensive application, the program conducts a significant amount of computation to solve the equation with only local variables referenced.

The program, Bodytrack, tracks the 3D pose of a human body through an image sequence using multiple cameras. The algorithm uses an annealed particle filter to track the body pose using edges and foreground segmentation as image features, based on a 10 segment 3D kinematic tree body model. We run the Bodytrack with simsmall input set which has 4 cameras, 1 frame, 1,000 particles, 5 annealing Layers. The program has mainly two parallelized kernels CreateEdgeMap and CalcWeights. CreateEdgeMap, resulting in a 2.5% speedup. CalcWeights resulting in a 1% speedup.

The Freqmine application employs an array-based version of the Frequent Pattern growth method for Frequent Itemset Mining (FIMI). In order to pin threads to the fixed cores, we use the non-dynamic scheduled

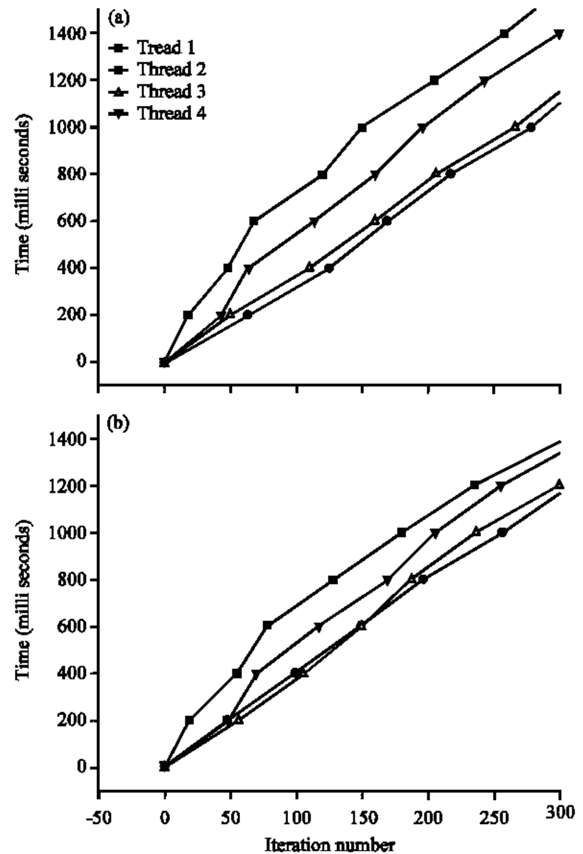

Fig. 5(a-b): Freqmine(fpgrow) run on four cores (a) Runtime behavior of the baseline and (b) Runtime behavior after applying critical thread guided cache management

version. Overall, the performance benefit correlates with imbalance level, performance benefit achieved 6%. In Fig. 5, we could find out that after applying critical thread guided cache management, though the performance of non-critical thread 2 and thread 3 degraded, the critical thread is accelerated, so the execution time of application decrease.

DISCUSSION

There has been a significant amount of research in shared CMP caches (Zhibin *et al*., 2012; Hameed *et al*., 2012), however, the underlying capacity control mechanisms used in these proposals are often based solely on spatial partitioning of shared caches, especially in architecture design and process/thread scheduling in OS. This study is distinctive in that it is shared cache capacity management that takes into account not only spatial allocation of shared caches but also temporal characteristics of cached data, furthermore, it study of shared cache capacity management considering thread behaviors in parallel applications, by the means that dynamically detect the critical thread.

In architecture research, many studies i.e., Wang *et al.* (2010), Chang and Sohi (2007) and Jyothi *et al.* (2007) have proposed various methods to design shared cache to balance the destructive and constructive effects of cache sharing. These studies, mainly focus on the hardware design, also containing some examination of the influence of shared cache. Their measurements are covered limited factors on the program or OS levels. Bitirgen *et al.* (2008) uses cache partitioning as its fundamental mechanism for the shared cache while coordinating other on-chip shared resources: cache bandwidth, shared caches and power budgets. Jaleel *et al.* (2010) proposed RRIP which modifies cache replacement policies based on cache line reuse interval prediction. While this work is similar to ours, critical thread guided fine-grained adaptive capacity management is distinct in its application priority handling, treating the critical thread and no-critical thread separately.

In OS research, the main focus on shared cache has been workload co-scheduling including thread clustering. Many workload co-scheduling studies (Su *et al*., 2012; Samih *et al*., 2011; Jiang *et al*., 2010; Tian *et al*., 2009; Zhang *et al*., 2013) are on multiprogramming environments, attempting to alleviate shared cache contention by placing different workloads appropriately. Some of them include parallel programs in the job set (Wei *et al*., 2009), but the main focus is on cache contention between different jobs rather than the effective of shared cache on parallel threads.

Many works proposed the method of finding out the critical thread in parallel application. It can be done with different approaches. Static on-line profiling information can be used to inform the OS about the iteration borders. Some authors build runtime libraries that dynamically detect the percentage of load balance per application (Liu *et al*., 2005). Some authors consider statistics last level cache miss ratio to find the critical thread (Bhattacharjee and Martonosi, 2009), but this method is not accurate as our scheme. Other schemes based on analyzing performance counters are also useful to measure these parallel iterations (Chang and Sohi, 2007). In study of Cai *et al.* (2008), introduction of checkpoints of the parallel application to find out unbalanced points in a parallel loop, but it's only work on SMT to balance the different threads.

CONCLUSION

Our critical thread guided fine-grained adaptive capacity management which have a mechanism detects the critical thread dynamically in a CMP, determining whether it to trigger cache allocation policy or keep using LRU. When triggered, our algorithms assign more cache space to the slowest threads of the application. We have suggested that these assignments should be done at the end of parallel loop, as coarse granularities may have problems with workloads that exhibit large IPC variations during their execution. Our proposed method can improve the overall performance of parallel region effectively with its fine-grained capacity allocation guided by detecting the critical thread accurately; Experiments have shown that performance improvement for the Computer Vision and Data Mining applications, ranging from 1-6%.

However, a more complicated scheme can be designed to take into account the data sharing characteristics: shared versus private data in a parallel application. For example, assigning a longer decay interval to shared data can help reduce the eviction rate of shared L2 cache blocks and their corresponding L1 cache blocks. As a result, the amount of coherence messages in the interconnection network between L1 and L2 caches can be significantly reduced. Furthermore, this can also reduce the overall application cache miss rate. Although the current our project does not take use of the data sharing property in parallel applications, this opportunity remains for future work.

ACKNOWLEDGMENT

This study was supported by the National Natural Science Foundation of China (Grant No.61173037).

REFERENCES

Bhattacharjee, A. and M. Martonosi, 2009. Thread criticality predictors for dynamic performance, power and resource management in chip multiprocessors. Proceedings of the 36th Annual International Symposium on Computer Architecture, June 20 - 24, 2009, Austin, TX, USA, pp: 290-301.

Bienia, C., S. Kumar and K. Li, 2008. PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. Proceedings of the IEEE International Symposium on Workload Characterization, September 14-16, 2008, Seattle, WA, pp: 47-56.

Bitirgen, R., E. Ipek and J.F. Martinez, 2008. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture, November 8-12, 2008, Lake Como, pp: 318-329.

Boneti, C., F.J. Cazorla, R. Gioiosa, C.Y. Cher, A. Buyuktosunoglu and M. Valero, 2008a. Software-controlled priority characterization of POWER5 processor. Proceedings of the 35th Annual International Symposium on Computer Architecture, June 21 - 25, 2008, Beijing, China, pp: 415-426.

Boneti, C., R. Gioiosa, F.J. Cazorla, J. Corbalan, J. Labarta and M. Valero, 2008b. Balancing HPC applications through smart allocation of resources in MT processors. Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008. April 14-18, 2008, Miami, FL, pp: 1-12.

Cai, Q., J. Gonzalez, R. Rakvic, G. Magklis, P. Chaparro and A. Gonzalez, 2008. Meeting points: Using thread criticality to adapt multicore hardware to parallel regions. Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, October 25-29, 2008, Toronto, Ontario, Canada, pp: 240-249.

Chang, J. and G.S. Sohi, 2007. Cooperative cache partitioning for chip multiprocessors. Proceedings of the 21st Annual International Conference on Supercomputing, June 17-21, 2007, New York, pp: 242-252.

Hameed, F., L. Bauer and J. Henkel, 2012. Dynamic cache management in multi-core architectures through run-time adaptation. Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE), March 12-16, 2012, Dresden, pp: 485-490.

Hu, Z., S. Kaxiras and M. Martonosi, 2002. Let caches decay: Reducing leakage energy via exploitation of cache generational behavior. ACM Trans. Comput. Syst., 20: 161-190.

Jaleel, A., K.B. Theobald, S.C. Steely, Jr. and J. Emer, 2010. High performance cache replacement using re-reference interval prediction (RRIP). Proceedings of the 37th Annual International Symposium on Computer Architecture, June 19-23, 2010, New York, NY, USA, pp: 60-71.

Jiang, Y., K. Tian and X. Shen, 2010. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilation, January 25-27, 2010, New York, pp: 201-215.

Jyothi, V.L. and S.K. Srivatsa, 2007. Proportional share resource scheduler with processor affinity in multiprocessor systems. Inform. Technol. J., 6: 561-566.

Liu, C., A. Sivasubramamiam, M.T. Kandemir and M.J. Irwin, 2005. Exploiting barriers to optimize power consumption of CMPs. Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium, 2005, April 04-08, 2005, Denver, Colorado, USA, pp: 5a.

Samih, A., Y. Solihin and A. Krishna, 2011. Evaluating placement policies for managing capacity sharing in CMP architectures with private caches. ACM Trans. Archit. Code Optim., 8: 1-23.

Su, C.Y., D. Li, D.S. Nikolopoulos, M. Grove, K. Cameron and B.R. de Supinski, 2012. Critical path-based thread placement for NUMA systems. ACM Sigmetrics Perform. Evaluat. Rev., 40: 106-112.

Tian, K., Y. Jiang and X. Shen, 2009. A study on optimally co-scheduling jobs of different lengths on chip multiprocessors. Proceedings of the 6th ACM Computing Frontiers, May 18 - 20, 2009, New York, pp: 41-50.

Wang, X., Z. Ji, C. Fu and M. Hu, 2010. A review of transactional memory in multicore processors. Inform. Technol. J., 9: 192-200.

Wei, G., H. Liu and M. Xie, 2009. Clustering large spatial data with local-density and its application. Inform. Technol. J., 8: 476-485.

Zhang, Z., S. Li and J. Zhou, 2013. Estimate load-dependent service demand for modern CPU. Inform. Technol. J., 12: 632-639.

Zhibin, H., Z. MingFa, X. Limin, R. Li and D. Yi, 2012. A novel online measure of cache utility efficiency in chip multiprocessor. Proceedings of the 11th International Symposium on Distributed Computing and Applications to Business, Engineering and Science, October 19-22, 2012, Guilin, pp: 72-76.