ITC 3/47
Journal of Information Technology and Control
Vol. 47 / No. 3 / 2018
pp. 588-608
DOI 10.5755/j01.itc.47.3.18433

On Improving Efficiency and Utilization of Last Level Cache in Multicore Systems

Received 2017/06/22 | Accepted after revision 2018/08/08

http://dx.doi.org/10.5755/j01.itc.47.3.18433

On Improving Efficiency and Utilization of Last Level Cache in Multicore Systems

Yumna Zahid, Hina Khurshid, Zulfiqar A. Memon

Department of Computer Science, National University of Computer and Emerging Sciences (NUCES-FAST)
Karachi, Pakistan, yumna.zahid@gmail.com, hinakhurshid18@gmail.com, zulfiqar.memon@nu.edu.pk

Corresponding author: zulfiqar.memon@nu.edu.pk

With the increasing need of computational power the trend towards multicore processors is ubiquitous. The current on-chip architecture comprises multiple cores which usually share last level cache which can be physically distributed on chip. In order to provide system predictability, especially for a real time system where quality of service (QoS) depends on minimum miss rates and low worst case execution time (WCET) for applications running on different cores, efficient cache management techniques are required. Since memory hierarchy and its management is the key of overall system performance and access to off-chip memory for data consumes many clock cycles along with many units of power it is important to restrict the off-chip access and provide the optimum solution for the on-chip access. To increase performance and energy efficiency, various techniques are proposed. This article aims to provide the researchers with the state-of-the-art critical review of the various approaches that focus on data replication and cache partitioning techniques for L3 cache. The existing literature is presented through several classifications based on appropriate design and algorithm. Maintaining energy efficient system is a crucial challenge for multicore processors. We have discussed various techniques which address upscaling performance without compromising on energy efficiency. Lastly, different literature work is discussed where authors evaluate cache and/or various processors for high performance applications such as bioinformatics, image and video processing, IOT applications and applications using DSP processors.

KEYWORDS: last level cache, multicore, data replication, cache partitioning, cache management, energy efficiency, high-performance applications.

1 Introduction

As the computational power and large amounts of data processing are increasingly in demand, on-chip multicore optimal performance becomes crucial. Modern multicore architectures support hierarchi-

cal memory organization; this includes private L1 and L2 caches for each individual core, L3 shared on-chip cache, and off-chip DRAM. Last-level Cache (LLC) or L3 cache is an integral component where data stored are easily accessed by many cores concurrently. LLC allows low latency access rate for data with high temporal and spatial locality compared to off-chip memory data retrieval. However, multicore performance can easily degrade due to large evictions of data from LLC and interference caused by cores performing on the same cache lines. Although L1 and L2 cache eliminates the problem of cache contention but offers limited space for data storage. The resultant interference and data eviction can lead to highly variable performance that can be detrimental to the system quality of service (QoS). Energy efficiency is another vital aspect to consider in multicore systems. Higher energy requirement not only is an unsustainable practice but can also generate poor system predictability. Thereby, it is imperative to develop countermeasures to efficiently utilize LLC for improved performance and energy consumption. This article provides two categories of measurements for improved LLC utilization: data replication and partitioning techniques applied on LLC. Table 1 shows the comparative analysis of the various techniques employed to optimize the L3 cache using Data Replication and Cache Partitioning.

Data replication mechanism is essential for maintaining high cache hit rates and data locality by replicating cache lines on LLC close to the requesting core [39]. Proposed data replication protocol offers many advantages. The protocol will decrease energy consumption (resulting in low heat generation) and memory latency by replicating cache lines which will be reused frequently in the last level cache of the requesting core. Moreover, with the help of a classifier, which will be adjusted on runtime at the granularity of cache lines, the protocol will balance data locality on caches and off-chip miss rates. This protocol will allow coherence complexity which will be similar to classic coherence protocol. Data replicas will be allowed on the last level cache of core where the request will be made. Furthermore, coherence complexity will also be allowed. This means if a miss occurs in the L1 cache and as a result, the last level cache will be searched for requesting data; data are invalidated on the local cache of a core.

Partitioning techniques increase process independence, reduce interference among jobs running on different cores concurrently, and hence utilize processor's capacity efficiently. Many methods have been implemented or proposed both in software and hardware to address issues regarding partitioning. This paper addresses some cache partitioning techniques most common being page coloring which brings an improvement to way-partitioning by allocating different colors to pages assigned to tasks [25]. Page coloring reduces chances of overlapping of L3 cache space among processors. This article describes how page coloring is employed. OS scheduling algorithm also impacts contention among workloads. Two main scheduling algorithms, partitioned-based scheduling algorithm and global-based scheduling algorithm, are presented and evaluated against the performance of page coloring. Moving on, dynamic cache partitioning scheme known as COLORIS [73] is another methodology which addresses the issue of cache interference more finely by allowing dynamic re-coloring based on application phase transition [30]. This article discusses cache utilization with dynamic cache partitioning and the consequent overhead with re-partitioning. Both of the above-mentioned techniques are still coarse-grained, that is to say, addresses partitioning at the set of blocks. Vantage, another method, implements fine-grained allocations to processes on multiple cores by partitioning at cache lines [61]. It proposes high associativity with line placement in LLC. There is increased isolation brought on by partitioning most of the cache rather than all of it. This way, when workloads assigned to partitions require more capacity than allocated, can borrow space from the un-partitioned region and so reduce eviction of other cache lines. This article uses statistical analysis to evaluate Vantage. In order to exploit heterogeneity in spatial locality among workloads, a two-dimensional approach called Spatial Locality-aware Cache Partitioning (SLCP) [26] was proposed. It claims to modify cache line sizes as well as allocating capacity [36]. It calculates capacity requirement in terms of the temporal and spatial locality at run-time for each individual task. Software techniques, as described, alleviate cache interference to an extent but require cache locking, implemented in hardware, to fully isolate tasks among cores.

The rest of the paper is organized as follows. Section

2 provides an insight into multicore architecture and memory hierarchy. It also elaborates how energy plays an important role when designing such multicore platforms. Section 3 focuses on Data replication techniques. Section 4 discusses shared cache partitioning techniques. In Section 5, energy efficient implementation with respect to both data replication and partitioning techniques on LLC is presented. Section 6 gives some insight on how LLC design plays a crucial role in High-Performance Computing. Moreover, Section 7 enlist some of the main limitations to *Data Replication* or *Cache Partitioning* techniques. Lastly, Section 8 concludes the research paper.

2. Background

In order to increase the processor's performance which can be affected by various reasons including memory latency computer architects proposed the idea of number of cores on a single processor chip. Multi-core processors do various tasks like multithreading, multitasking, security and other physical checks which results in higher heat generation. This in turn will raise other issues like scalability and power constraints among multi-core network communication. Applications used multithreading on multicores to get faster operations. In order to improve performance and energy efficiency good scalability for multicore and assurance of single core performance is important [39].

Memory hierarchy efficiency is directly proportional to access latencies of private and shared caches and their hit rates. Much of the die area of the processor occupied by the Last Level caches are expected to hold quite a few megabytes of data. The shared Last level cache, which provides megabytes to multicore, is hard to manage since it requires cache coherency protocols on architectural level. These protocols utilize data locality and scalability of directory to get data faster from local caches in single chip multicore organizations but, on other hand, data continue to displace on private caches. Private caches also suffer from capacity limitation due to frequent communication between cores over data [40]. Capacity limitation is also one of the reasons for cache miss rates [5]. In order to reduce these miss rates several techniques have been proposed so far like larger block-size, Instruction pre-fetching, data pre-fetching, higher associativity, controlled pre-fetching, compiler pre-fetching and victim caches. These techniques reduce miss rates, however cache performance is not enough due to often miss penalties in case of cache miss. Here introduced the idea of multi-level caches along with techniques like *priority to read miss over write miss*, *non-blocking caches*, *critical word first* etc. Introduction of multi-level cache significantly improves performance of overall system as compared to single cache organization. Later, when multi core was introduced with already proposed techniques, the data locality and off chip miss rates were balanced by Last level cache organization. Private LLC organization, on the other hand, had high off chip miss rate. Non-Uniform cache access is a result of shared LLC organization that effects chip locality but their off chip miss rates are low since cache lines are not replicated [37].

Multicore processor improves parallelism and performance as compared to single core processors, but it comes with complexities like coherency, memory consistency and synchronization issues. Many solutions have been proposed so far in order to reduce complexities and achieve maximum performance from multicore architecture. To reduce coherence complexity, data migration and data replication, enforced on cache using snoopy or directory protocol. However, snoopy protocol does not show significant performance when the number of cores increased more than eight cores per processor. Here directory protocol takes lead with contrast to various cache coherency protocols like MSI, MESI, MOSI, MOESI, MESIF, MERSI, DragonFly and FireFly.

Multicore processors have their own private (L1 and/or L2) and shared cache (L3 and/or L4 also known as Last Level Cache) where directory is stored on each LLC of core that is also shared with cores. Directory keeps track of the metadata on each block and of which cache block holds what data along with its status.

3. Data Replication Techniques

This section will elaborate on the various data replication techniques employed in LLC to enhance the efficiency of multicore processors. Multiple approaches are discussed as proposed by different researchers.

3.1. Data Replication on Last Level Cache

Baseline multi-core system contains network controller for communication between cores and maintaining network traffic. Each core contains private cache along with cache lines, shared cache (Last level cache) with directory to maintain coherency among caches. In classic approach of MESI (Modified-Exclusive-Shared-Invalid) protocol, if one private cache L1 of core A holds data and another private cache of other core B requests for the same data, then it sends request on network [10]. If requested data in private cache of Core A are valid and the respective core is free to respond back, then Core B is given the requested data and data block is marked with Shared state; otherwise core B needs to wait. Meanwhile, if another core C requests

Table 1
Comparative Analysis of L3 Cache Optimizing Techniques

	Approach	Techniques	Overheads	Performance
Data Replication	On Last level cache	Data replacement using Locality Classifier, MESI Protocol	Storage overhead, expensive coherence protocol, more network traffic	14% to 21% improvement on different parallel benchmarks
	On CMP-NuRAPID	Data replication near to requestor core using In-Situ communication, MESIC protocol	Cache capacity overhead, communication overhead for every write	13% to 28% improvement over a shared cache and 8% improvement over private caches
	Tag replication	Replication of cache tag into TRB, Selective TRB	17.2% energy and 15.6% area overheads	Selective-TRB achieves AWR rate of 97.4% for dirty cache-lines. Selective-TRB-EWB achieves AWR rate of 100% and 0 DOR-AVF
	On Clusters	Block replacement in distributed caches, rotational interleaving	Hardware overhead	Average 14%, best 32% , ideal cache design 5%
Cache Partitioning	On Real-Time Schedulers	Page coloring, Partitioned-EDF and Global-EDF scheduling employed	OS overhead (scheduling, context switching, release, IPI latency, and tick counting)	Safe HRT bounds for WSSs of 32 and 64 KB for both G-EDF and P-EDF, and for WSS of 128 KB for P-EDF
	Using COLORIS Framework	Static cache partitioning and dynamic cache partitioning using page coloring	Re-partitioning of pages overhead	Up to 39% decrease in co-running workload interference
	Vantage Cache Partitioning Scheme	Fine-grained partitioning for skew-associative caches and zchaces. Isolation techniques using managed and un-managed regions. Churn-based management	1.5% state overhead	Improves throughput by 98%, 8% on average(up to 20%), using a 4-way cache
	Spatial Locality-Driven Cache Partitioning (SLCP)	Using spatial and temporal locality of workloads at runtime. Set-associative tag structures using auxiliary tag directories (ATD)	Tag storage required. Hardware overhead of online locality monitoring	18.2% and 18.4% higher IPC throughput using LLC prefetcher. 6.9% to 12.5% energy consumption reduction
	Semi-Partitioned Hard-Real-Time Scheduling	Semi-partitioned scheduling using cache-locking and locked cache migration	Task migration overhead	37.31% increase in utilization & 81.36% density increase

for the same data, then it needs to wait too in order to get requested data. This phenomenon effects processor performance while cores are in waiting state. It also increases network traffic since cores need to check repeatedly if data are available [48].

To overcome the issue, many researchers have proposed replication techniques. Kurian et al. [39] have come up with the idea of copying data into requestor LLC slice so that if private cache needs data it should be available to its own shared LLC. This mechanism will track and classify reuse of each cache line of last level cache with the help of classifier who will track down the locality information of the cores.

3.1.1. Methodology

Authors classify each core into either *replica sharer* or *non-replica sharer*. Initially, all cores will be *non-replica sharer* since there is no replica made so far. Each core, *either replica sharer* or *non-replica sharer*, will contain home *reuse counter* and *replica reuse counter* along with *replica threshold*.

On a read miss, request will be sent to home location and data will be replicated on private and LLC cache if *replication bit* mode is true (*replication bit* is set to true when home reuse counter gets to *replication threshold*).

Replication bit mode and *home reuse* counter (tracks the number of times home location is accessed by specific core for data) is initially set to zero (0). When request for data comes to home location, *reuse counter* is incremented by one. If *home reuse* counter is smaller than the threshold value, data will not be replicated and only requested data will be given to private cache of requester core.

When a miss occurs for write request, directory protocol checks local LLC for replicated data.

1. If replica is not found, request will be sent to home location. On each access request, home reuse counter will be incremented. As soon as home reuse counter reaches the threshold value, replica will be created for requester core; otherwise requested data will be directly given to requester core.
2. If replica is found at LLC location in *Modified (M)* or *Exclusive (E)* state, data are given to private cache and replica counter will be incremented.
3. If replica in LLC block is in *Shared (S)* state, then directory will first invalidate all replicas in LLC and data in private cache before updating and home reuse counter will be set to 0 for all non-replica sharer except writer.

By far Least Recently Used (LRU) protocol is considered to be most successful block replacement protocol but it still has a room for improvement. Various authors have proposed modified versions of LRU [18]. Kurian et al. [39] proposed locality classifier which with the help of the locality information identifies if core will remain replica sharer or not. It helps to identify when any invalidation or eviction request comes to core.

3.1.2. Evaluation

By reviewing ideas for data replication it has been observed that data replication has a fault of storage overhead since data replication is placed in cache. For example, lets suppose 16kb data be stored in local private cache of 64kb, whereas its replication copy (16kb) is stored in its local LLC of 128kb. Replication space of the same size is also filled in home location LLC and home location private cache. For the 16kb data, additional storage overhead (considering only replication data size in LLC of both cores) is 32kb (almost double the size in this example). Moreover, in order to keep track the replicated data and to maintain the coherency between them, a directory is also needed which will take its own storage capacity. On the other hand, coherency protocol capacity also increases due to the addition of separate tags and counters of the directory protocol to maintain coherency in private caches of the replicated data. Additionally, classifier needs to update locality information each time along with the replicated core sharer list [6]. This makes the directory an expensive solution as compared to previously proposed coherency protocols.

Network traffic increases when communication between caches and LLC occurs for the acknowledgment of invalidation and eviction [39]. These signals are actually the acknowledgment messages, used to send invalidation and eviction information along with home reuse counter and replica reuse counter.

3.2. Data Replication on CMP-NuRAPID

To improve the performance of Chip MultiProcessors (CMP), various designers have combined design metrics with constraints in order to optimize performance [42, 56]. Many researchers have proposed different approaches [68, 72, 74] to optimize CMP per-

formance. Chishti et al. [15] proposed data replication ideas on CMP-NuRAPID that replication should be made near to requester core and it should only take place for read requests. They replicated data on requester core if only read requests are made from core. They controlled replication by avoiding replicating data for read-write requests.

3.2.1. Methodology

The authors in [15] have explored the possibilities of achieving fast access and enough capacity in shared and private cache by proposing the following ideas:

1. To allocate copies (for read-only requests) close to the requesting core, so that access should be fast. Controlled-replication is required to minimize capacity issue raised by replicating copies near to the requesting core.
2. Since off-chip communication is slower than on-chip communication, in-situ communication was proposed to have fast access to data.
3. The authors have proposed the idea of *capacity-stealing* from neighboring cache, since communication between the neighbors in CMP is not expensive. This provides dynamic customization of on-chip capacity.

The authors also exploits data arrays, tag arrays and pointers in CMP-NuRAPID architecture to execute proposed ideas. In order to achieve replication for read-only requests MESI protocol was modified. A new state was added in already existing MESI protocol so that in-situ communication can also be achieved. This new protocol was named as MESIC (Modified, Exclusive, Shared, Invalidate, Communication).

3.2.2. Evaluation

Chishti et al. [15] analyzed the performance metric for controlled replication and In-situ communication individually and together with metric parameters; multithread workloads and multiprogrammed workloads, respectively [9]. Results show that proposed techniques significantly decrease capacity and Read-only sharing misses. On average, CMP-NuRAPID achieved better performance over shared cache by 13% and for private caches by 8%.

3.3. Tag Replication Along with Cache Line

Replication on data level still lacks significant outcome and provides a room for improvement. To keep this in consideration, Wang et al. [70] proposed an idea to replicate cache tag along with cache data. They utilize data locality information to replicate most recently accessed cache tag into buffer named *Tag Replication Buffer (TRB)*.

3.3.1. Methodology

Tag replica is created when data are fetched into private cache and there is no tag entry of it in tag replica buffer and second when data are fetched into private cache and its entry is added into tag array. In order to improve security and reliability of TRB, [70] further proposed *Selective-TRB*. This scheme only works for dirty cache lines and replicates only dirty cache line data. The authors in [70] further exploit LRU replacement policy to propose their own modified LRU policy with the help of FIFO.

3.3.2. Evaluation

Wang et al. [70] exploited memory access locality to design TRB and to increase reliability in TRB. The authors also proposed *Selective-TRB* which displayed significant performance rate in contrast with other proposed work for data replication and smaller overhead since, these techniques works on dirty cache lines. TRB shows performance improvement of almost 90% when tags are accessed via tag buffer (keeping tags replica). Their Selective tags replication technique with modified replacement policy LRU+FIFO shows improvement of 97% when performed over tags of dirty cache data.

3.4. Replication on Cluster Level

Hardavellas et al. [28] proposes replication of data on clusters for reactive-NUCA architecture. Cluster level replica is defined as a number of cores where maximum one replica should be present on a cache.

3.4.1. Methodology

During analysis the authors in [28] observed that different characteristics were exhibited, when the cache was accessed both for instructions and data (in both private and shared caches), which in result leads to the implementation of not only different data migration policies but also different replication and placement policies [7]. By exploiting this observation, the authors designed Reactive-NUCA for block placement in distributed caches with lower overhead and latency. Architecture places

blocks at the appropriate location of cache by reacting to class of cache access. The proposed technique intelligently works with O.S to minimize coherency issues and support data placement, migration and replication policy without using any external protocol.

3.4.2. Evaluation

The size of cluster was increased by adding as many cores as possible, resulting in increasing hit latency on last level cache but reduced data locality and last level miss rates. Since miss rate on the last level did not improved and also due to the existing clustering, each location needs to search whether the data is present in L1 cache or not, causing not only delays in network but also exhibit drastic network performance.

However, results show that RNUCA displayed performance stability and improvement on average by 14% for private cache designs, 6% for shared and 5% for ideal cache design. Their maximum achievement was 32%.

4. Cache Partitioning Techniques

This section lays out a comparative analysis of various methodologies proposed to isolate workloads in LLC and to help alleviate interference between co-running processes on multicore processors.

4.1. Page Coloring Cache Partitioning

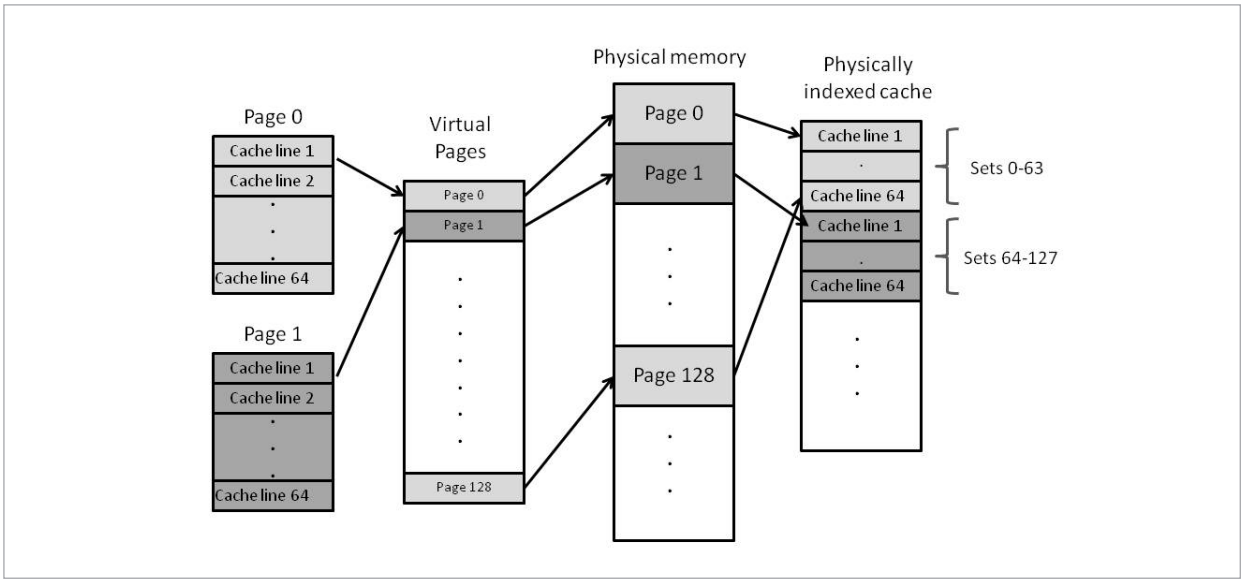
Page coloring cache partitioning technique, as proposed by Gracioli and Frohlich [25], allows isolation in task workloads in multicore processors by assigning different colors to individual tasks [27]. This section discusses the methodology used and the experiments carried out on various working set sizes (WSS) using either Partitioned-based scheduling algorithm (Partitioned-EDF) or Global-based scheduling algorithm (Global- EDF), and its evaluation.

4.1.1. Methodology

Page coloring is a software-based partitioning technique designed to reduce cache interference caused when a task on one core evicts L3 cache line belonging to a (possibly) preempted task of the same core or another core. Page coloring utilizes the virtual to physical page address translations in a set-associative indexed cache [66]. Using an 8MB shared 16-way set associative cache with 64-bytes per line, we are provided with 213 sets in the cache (8MB/16 ways x 1 way/64 B). The first 6 bits in the cache address relate to words in the cache line, the next 13 bits access a set, and the next 13 bits define a line from one of the 16 ways.

In page coloring, colors are assigned to each page such as color 0 to page 0, color 1 to page 1 and so on. The colors are repeated after reaching the maximum col-

Figure 1
Mapping of physical pages to cache locations [24]



or, calculated as (cache size / number of ways / page size). Hence, in our example, page 128 maps to the same color as page 0. Figure 1 shows how the actual pages are mapped to each cache set.

Scheduling algorithms play an important part in the performance of partitioning schemes. The two algorithms discussed in Page Coloring Methodology are global-based scheduling and partitioned-based scheduling. Global Earliest-Deadline-First (G-EDF) [24] is a type of global scheduling algorithm where the OS defines one ready queue of tasks to be distributed among the available processors. When a job is preempted from one core, it can be migrated to another processor for resumption. On the other hand, in a Partitioned-EDF (P-EDF), a partitioning heuristic statically assigns tasks into available processors where they are executed and no migration takes place for preemptive tasks. G-EDF is optimal for mixed-criticality levels co-exist in the same system.

4.1.2. Evaluation

The experiment was carried out on an Intel i7-2600 processor [62]. The super colors were assigned to the number of tasks in each set with two additional colors for an uncolored heap and another one for the OS. The super color is calculated when the number of colors defined for tasks are less than the maximum number of colors, as given by (1).

$$Supercolor = pagecolor\%max.num.ofcolors. \quad (1)$$

The experiment was carried out with the following scenarios by evaluating page coloring in terms of OS scheduling algorithms on the Worst-Case Execution Time (WCET) of tasks.

- **S1:** OS and each task allocate data from a different super color.
- **S2:** Each task allocates data from a different super color. OS allocates data from a non-colored and sequential heap. This creates interference between data allocated by OS and the data of each task, because OS can access a cache line of any color.
- **S3:** Each task allocates data from the same super color. OS allocates data from a different super color.

Page coloring cache partitioning increases the system predictability by meeting deadlines [67]. It achieves this by using isolation of workloads. In P-EDF, page coloring was effective up to 128 KB of WSS and in G-EDF up to 64 KB due to some interference caused during migration. Inter-core communication is reduced by cache coherency protocol since all data are small enough to be able to fit in L2 cache and invalidations are minimized in L3. For larger WSS, the P-EDF is effective since partitioning of the L3 cache isolates and hence avoids contention between tasks running on different cores simultaneously.

4.2. Coloris: Dynamic Page Coloring

Due to the dynamic nature of applications, frequent repartitions of L3 cache become crucial for cache utilization. Static partitioning as discussed in Section 4.1 fails to address re-coloring repartitioning based on application phase change. Ye et al. [73] came up with COLORIS (COLOR ISolation), which is a framework implemented to dynamically repartition cache while maintaining fairness or Qos.

Re-coloring is a tedious task that can incur substantial overhead. It requires allocation of page frames, copying pages from memory and freeing old page frames when necessary. There can also be chances of naive allocation of colors among applications in such a way as to increase contention among tasks and consequent thrashing. A simple example would be a 2-core, 6-page color system with two running processes (P1, P2) and one ready process (P3). The page colors allotted to these three processes might be 1, 2, 3, 4, 5, 6 and 1, 2, 3, respectively [35]. Now, if process P2 needs to be preempted and replaced with process P3 to run on one core while P1 continues to run on the other, two processes will increase contention drastically for the same subset of space in the cache as the colors for both processes are the same. This would require re-coloring in an efficient manner.

This section will discuss its implementation details and the findings based on the experimental results.

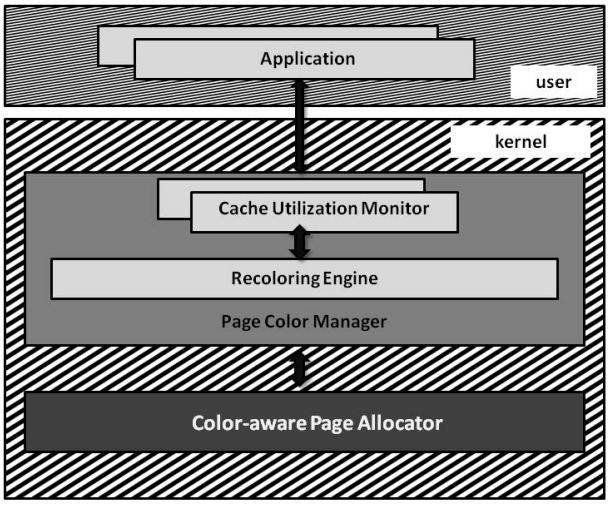
4.2.1. Methodology

Shown in Figure 2 is the architecture of COLORIS which comprises two major components: a Page Color Manager and a Color-aware Page Allocator.

The Color-aware Page Allocator has replaced the Linux memory allocator framework which involves overhead by frequently calling the Buddy System to refill the list of free pages of a specific color in the page frame cache [65]. In COLORIS, instead, a memory pool is utilized

for all page requests in a way that free pages, which are assigned the same color, are linked together and hence multiple lists are formed. The allocator, when requested for a page, collaborates with Page Color Manager to ascertain the colors assigned to the requesting process. The allocator then picks one of these colors in a round-robin manner and returns a page with that color from the memory pool. In the case of absence of that color another color is picked, also assigned to the process, or in the case of none of them present it contacts the Buddy System to populate the memory pool with new pages of various colors.

Figure 2
Architectural Overview of COLORIS [73]



The Page Color Manager, as already briefly explained, is a component in COLORIS responsible for assigning colors to processes via specific policies. In order to effectively utilize all the L3 cache space among cores, Color Manager adopts a more flexible scheme. It divides the cache into N contiguous sections with an equivalent number of cores. Then each section is assigned to the individual core of C/N page colors where C is the total number of colors available. In this way, N co-running processes have the advantage of fully utilizing cache capacity with enhanced isolation.

The static partitioning still incurs a large overhead when migration of processes needs to be performed in the case of load balancing. To overcome this problem and to exploit dynamic phase changing of the process, re-coloring is exercised. Page Color Manager makes online color assignment changes based on dynamic application behavior in such a way that if one process does not require the entire local cache section, some colors can be reclaimed; likewise, other processes can get more cache space by sharing colors from other sections [46]. This does, however, reinstate some interference but by sharing information on color utilization and scheduling processes using schedulers the chances of contention can be mitigated.

4.2.2. Evaluation

With respect to re-coloring, experiments were carried out using four benchmarks: *povary*, *tonto*, *omnetpp* and *gamess*. *omnetpp* is memory intensive with others having a smaller memory footprint in order for repartitioning to be possible. Multiple configurations were tested on a 32-bit Ubuntu 12.04 Linux OS with kernel version 3.8.8 using threshold values as shown in Table 2.

Table 2
Experimental Configuration [73]

Configuratons	LowThreshold(%)	HighThreshold(%)
C1	30	65
C2	30	75
C3	0	100
C4	(40, 0, 30, 40)	(80, 60, 65, 80)
C5	-	-
C6	30	75

These are global cache miss rate thresholds. Applications with miss rates higher than High Threshold require more cache space where as with applications having miss rates lower than Low Threshold can provide their local cache vacant space for repartitioning. In *C3*, re-coloring is not possible due to their threshold levels. *C5* was used for the special case in which every benchmark was executed using full cache space (for more details, please refer to the given reference [73]).

Using static cache partitioning, the miss rates of *omnetpp* reach 77.8%. Whereas, using re-coloring (*C1* and *C2*) the miss rates were contained within the range of threshold. This means QoS requirement was met. Instead of using system default QoS specification dependent upon uniform threshold values, independent QoS can be satisfied as shown in *C4* configuration. Here, Low Threshold was kept at 0 for *tonto* in order

to avoid taking its cache space and hence its miss rate was retained below 40%. This shows that individual QoS can be effectively guaranteed using COLORIS framework for repartitioning.

4.3. Vantage: Fine-Grain Cache Partitioning

Techniques, so far discussed in Cache Partitioning, are coarse-grained partitioning which allows sections in terms of sets (multiples of page size x cache way). This reduces associativity and cannot be extended to support a number of cores more than 4 without compromising QoS and isolation among partitions. Vantage, proposed by Sanchez and Kozyrakis [61], is a scalable and efficient technique which employs fine-grained cache partitioning. Vantage is capable of supporting a large number of partitions defined at cache line granularity for on chip multiprocessors (CMPs) with as many as 32 cores. It proposes to maintain high associativity and strong isolation. Vantage is applicable on caches with high associativity such as skew-associative caches [63] or zcaches [60] indexed with good hashing. It can work with set associative caches but with lower performance.

Vantage is derived from statistical analysis and not from empirical observation. It implements partitions partially most of the cache, instead of complete cache space. This way when partitions outgrow their allocations, they can take space from an unpartitioned cache instead of compromising space from other partitions which effectively reduces interference.

4.3.1. Methodology

Vantage uses soft partitioning which does not physically restrict line placement. It evicts lines using churn-based management. Churn-based Management uses insertion rates (insertions per unit of time) known as churns to match demotions of line per partitions. This is implemented by firstly dividing the cache into two logical regions – a managed and an unmanaged region, by using tags [49]. LRU (Least Recently Used) replacement algorithm is used to rank line irrespective of region. On eviction, lines from the unmanaged region have higher priority over lines in the managed region. Churn-based Management allots apertures per partition. This way replacement candidates below the assigned aperture are demoted first to the unmanaged region and either evicted or in the case of getting a hit are promoted back. Promotion and demotions are manipulated simply by using tags. Aperture (A) is the threshold value used to allow demotions on average rather than one candidate per eviction. This increases the associativity. For example, if A = 0.05, it will demote every candidate that is on the top 5% of eviction priorities. It thus keeps the insertion and demotion rates of each partition equivalent so that their sizes are approximately constant.

Since Vantage implements partitioning through the replacement process, this implicates for changes in the cache controller. The cache controller is given the target size of each partition which is set using external allocation policy (such as UCP [58]) and partition ID of each cache access. Each line is, thereby, tagged and on each replacement controller performs evictions from the unmanaged region, demotions or promotions from the managed regions based on access rates of cache lines.

4.3.2. Evaluation

The simulation is performed using an x86-64 simulator based on Pin [43] which models both small and large-scale CMPs. For small-scale configuration wherein simulations were carried out using a 4-core system, the performance is evaluated using a mixture of 350 workloads derived from different categories as listed in Table 3. There are 35 possible combinations of these four categories each forming a class. There

Table 3
Classification of SPEC CPU2006 workloads [61]

Insensitive(n)	perlbench, bwaves, gamess, gromacs, namd, gobmk, dealII, povray, calculix, hmmer
Cache-friendly (f)	bzip2, gcc, zeusmp, cactusADM, leslie3d, astar
Cache-fitting (t)	soplex, lbm, omnetpp, sphinx3, xalancbmk
Thrashing/streaming (s)	mcf, milc, GemsFDTD, libquantum

are 10 mixes per class with each application being randomly selected from the ones in its category yielding 350 workloads.

Vantage provides much larger improvements than either way-partitioning or Promotion-Insertion Pseudo-Partitioning (PIPP) giving a 6.2% geometric mean on average and up to 40% speedups. PIPP or way-partitioning shows worst-case performance of 29% and 22% respectively, as compared to 4% for Vantage. Partitioning adversely affects associativity for these workloads using way-partitioning or PIPP which establishes the importance of maintaining high associativity.

Vantage also shows higher performance in all workloads except one, i.e., an un-partitioned cache, which is a factor by which other configurations such as way-partitioning have an edge. However, in associativity sensitive workloads, Vantage has already outperformed both alternatives. Evaluation using zcache shows how high associativity in Vantage has allowed it to provide higher throughput.

4.4 Spatial Locality-Aware Cache Partitioning

Vantage does provide fine-grained partitioning design but is limited to: 1) caches with good hashing and associativity, and 2) it only exploits temporal locality [8]. Heterogeneity in the spatial locality is rarely used in the partitioning schemes discussed above. Spatial locality refers to block/line size which, if manipulated correctly, can reduce the capacity allocation among multiple cores and thereby provide drastic improvements in system performance. Gupta and Zhou [26] proposed Spatial Locality-Aware Cache Partitioning (SLCP), which leverages a two-dimensional optimization in cache partitioning wherein both block size and capacity is considered providing heterogeneous organization for various workloads.

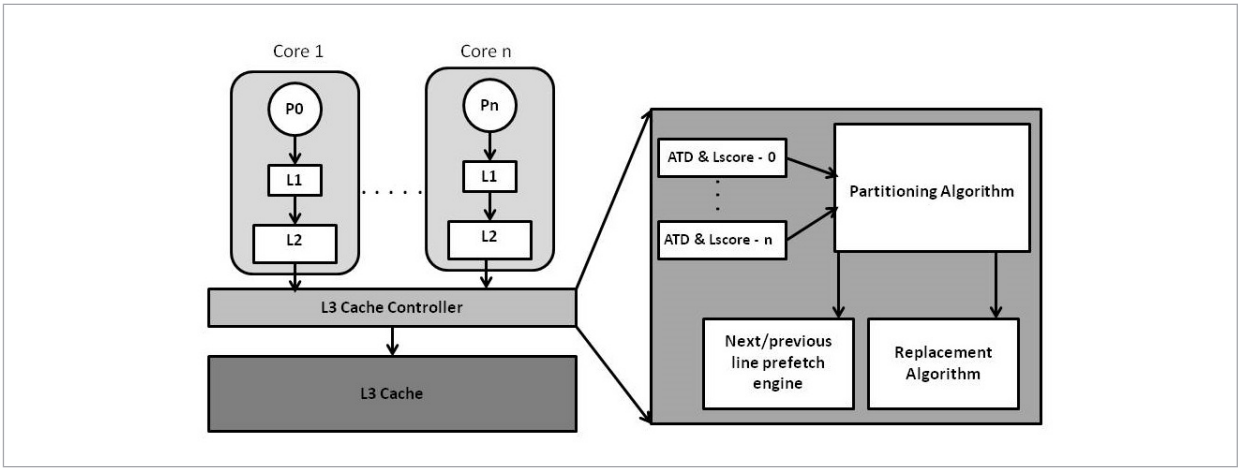
SLCP argues that, for memory intensive work, a larger block size would render a small overall cache capacity requirement. In order to achieve this goal, a unified approach is proposed in SLCP, known as *Locality Score* to measure both temporal locality and spatial locality at runtime and make amendments as to the optimal heterogeneous organization. *Locality Score LS(X, Y)* is a function that defines the hit rate based on the future window size *(X)* of a referenced address and the neighborhood size *(Y)* of a fully-associative cache with capacity of *(X * Y)*. *X* is essentially the reuse distance, whereas *Y* is the block/line size.

4.4.1 Methodology

SLCP online Locality-monitoring framework uses a hardware approach for estimating a two-dimensional locality (temporal and spatial) in cache access streams. Figure 3 shows an architectural view of the SLCP including processors and cache hierarchy where two things have been added: 1) the online locality monitor hardware, and 2) the logic for running the partitioning algorithm.

SLCP hardware is comprised of a few sets of auxiliary tag directories (ATD) [58] and locality score *(LScore)*

Figure 3
The SLCP Architecture for LLCs [26]



counters. Multiple ATDs and LScore counters are employed for different cache block sizes. All the ATDs have the same number of way as in the original cache so as to capture the locality information for different cache capacity. The LScore counters are used to record cache hits in the ATDs when varying number of ways and various block sizes are assigned.

Locality score counters are added to each core with the intent of maximizing the weighted sum of scores across all the co-scheduled benchmarks to be fed into the partitioning algorithm in SLCP.

The two-dimensional LScore array is based on *(L, K)*, where *L*thentry in the LScore–K counter maintains: 1) the number of cache hits for block size K, and 2) the capacity of $L^*(C0/\alpha)$, where *C*0 is the baseline cache capacity and $(C0/\alpha)$ is the capacity of one cache way using α as the associativity of LLC. This is reduced into a single-dimensional LScore vector *(L)* to leverage the lookahead algorithm [26]. The lookahead algorithm is used to determine the partition configuration only after a fixed amount of cycles. LScore counters are shifted, instead of resetting them completely, so that history is retained.

4.4.2. Evaluation

SLCP is implemented using an in-house execution-driven simulator. Baseline memory hierarchy configuration constitutes three levels of caches, private L1 and L2 caches and a shared non-inclusive L3 cache. SLCP is tested with the following five categories of 4-way multiprogrammed workloads: 4H, 3H1L, 2H2L, 1H3L and 4L. The 4H category has 4 benchmarks with high MPKI (Misses Per Kilo Instructions) and the 3H1L category has 3 benchmarks with high MPKI and 1 benchmark with low MPKI and so on.

SLCP shows an average of 18.2% (20.9%) performance improvement, highlighting the importance of leveraging spatial locality for partitioning. The performance improvements can be regarded as due to the joint optimizations of SLCP which assign less capacity to the benchmarks that can exploit spatial locality in limited cache capacity. It can also be perceived that multiprogrammed workloads with a higher number of high MPKI benchmarks tend to improve with SLCP. This can be awarded by assigning large cache block sizes which trigger higher hit rates as well as IPC improvements while at the same time also donate cache capacity to other benchmarks.

5. Energy Efficiency

Energy efficiency is a crucial consideration in multicore systems, especially in mobile devices where battery life can be adversely affected and devices tend to heat up leading to poor performance. Many architectures support disjoint execution of background and foreground applications to cater to high responsiveness which can increase energy tax in terms of battery life, power and capital expenditure. In multicore processors, the allocation of the core to applications and frequency of these cores are factors to consider while making energy efficient policies. Cache energy reduction techniques have been widely studied. In these techniques, turning off parts of the cache in order to reduce static energy is usually employed. Race-to-halt, a term coined, to indicate a scenario where additional cores used to speed up execution is liable to conserving energy by finishing up tasks quickly and consequently allowing the system to be in a low-power state [16]. This can, however, lead to counter-intuitive performance when memory-bound applications are run on a number of cores at high frequency and are in a waiting state for the data to be provided.

LLC spend a larger fraction of their energy in the form of leakage energy and hence need techniques which work by turning off a part of the cache to reduce the leakage energy consumption. These approaches based on the retentiveness of turned-off blocks, are broadly ramified into two techniques, namely state preserving and state destroying. Based on this, Li et al. [41] have compared the effectiveness of both techniques. They conclude that it is more cost effective to employ a state-destroying technique when fetching a missed block is not critical, compared to state-preserving technique. This is because state-destroying technique completely turns off the block and hence helps to conserve more energy.

For both state-preserving and state-destroying leakage control, architectural techniques make use of some well-known circuit-level mechanisms. Powell et al. [57] propose a circuit design named gated VDD, which facilitates state-destroying leakage control. This technique adds an extra transistor in the supply voltage path or ground path of the SRAM (static random access memory) cell. For reducing the leakage energy of the SRAM cell, this transistor is turned off and by stacking effect of the transistor, the leakage

current is reduced by orders of magnitude.

Several energy saving techniques are based on the generational nature of cache access, which implies that cache lines have a period of frequent use when they are first brought into the cache, and then have a period of dead time before they are evicted. Therefore, if a cache line has not been accessed for a certain number of cycles (called decay interval or update window), it indicates that the line has become dead and it can be put in low leakage mode for saving energy. Using this principle, Flautner et al. [21] proposed a drowsy-cache technique which puts the dead cache lines into low-power state-preserving mode. Similarly, Kaxiras et al. [34] proposed a decay cache technique which puts the dead cache lines into low-power state-destroying mode.

Several researchers have proposed improvements to the original decay-cache technique, but in all such proposals, the optimal value of the decay interval was varying with the applications. Zhou et al. [75] proposed a technique for dynamically adapting decay interval for each application. Their technique only turns off data and keeps tags alive. Using tags, their technique estimates the hypothetical miss rate, which would be there if all the data lines were active. Then, the aggressiveness of cache line turning off is controlled to make the actual miss rate to closely track the hypothetical miss rate. Abella et al. [1] keep track of the inter access time and the number of accesses for each cache line and use this to compute suitably decay time for each individual cache line.

Lu and Guo [44] proposed two dynamic voltage /frequency scheduling (DVFS) based algorithms: 1) pre-DVS, and 2) post-DVS, for multicore systems which employ fixed-priority scheduling with task splitting. The post-DVS, also known as DVFS after scheduling, works like a conventional DVS for fixed-priority scheduling. It allocates just the right amount of frequency to sub-tasks which are split, after scheduling in a way that tasks which are performed first are executed quickly so the leading ones get enough frequency to meet the deadlines. This conserves energy while meeting timing constraints on synchronization for scheduling with task-splitting. Moreover, pre-DVS performs pre scheduling frequency evaluation of tasks. This assumes prolonged execution times for scheduling of tasks so that all tasks are completed within the required time limit so energy is saved, achieving more energy conservation than post-DVS. It manages this by determining the total utilization of task-set and number of available cores so that every task is divided equally among all the cores and the energy is maximally conserved.

Another methodology, as proposed by Xu et al. [71], refers to minimizing energy consumption in multicore platform for parallel tasks. It takes a practical approach by considering processors with discrete modes of operations and have timing constraints. The algorithm discussed operate on either rigid task, which have fixed parallelism or moldable task whose parallelism can only be decided at run time. For both types of tasks, first the problem is formulated as a 0-1 Integer Linear Program (0-1 ILP) and then either a two-step (rigid tasks) or a three-step (moldable tasks) heuristic is applied. In the first case, the heuristic schedules tasks using a level-packing algorithm and then it decides upon another step to determine the level of frequency required with minimum energy consumption. Similarly, in moldable tasks, the third step also addresses the level of parallelism required for each individual task. Based on simulation results, the heuristics energy consumption is almost equal to that of 0-1 ILPs.

Chen at al. [13] combined both DVFS and Dynamic Power Management (DPM) to address the energy consumption issue with multicore systems. Their approach is based on Mixed Integer Linear Programming (MILP), optimizes both DVFS and DPM for applications composed of a set of tasks. It uses acylic graphs (DAG) to represent their precedence levels while mapping them on multicore processors. The energy model considers varied sources of power consumption for a set of discrete frequencies, and also the time/energy overhead. The algorithm is used to determine the optimal time-triggered non-preemptive schedule and execution frequency of tasks in an application, and in doing so reduces total energy consumed in MPSoCs.

Mittal and Zhang [51] used dynamic cache reconfiguration techniques for cache leakage energy saving. The caches are configured to conserve maximum energy and keep performance sensitivity bounded. They tested a large number of potential configurations using low-overhead and micro-architecture components with easy integration on multicore chips. The methodology makes certain that energy is uniformly conserved throughout the system, outperforming other

comparable methodologies used in high end embedded, desktop, servers and other multitasking systems.

6. Last Level Cache in High-Performance Applications

Introduction of caches in third generation computer architecture solved the problem of slowness and expensiveness of main memory in early decades of computing. As time passed by, level of caches were introduced to achieve higher CPU performance. Current growth in the area of high speed performance applications like physics simulations e.g. ALE3D, chemistry or biology applications e.g. IBM sequoia; require high performance of processors. As time passes, many techniques are introduced in order to optimize processor performance like pipelining, higher clock speed, parallelism, branch prediction and even number of transistors are added per chip. These traditional approaches, however, bring some limitations or challenges like memory latency or power dissipation.

To hide memory latency, many multithreaded techniques have been introduced and adopted by processors like Intel Xeon family processors, Atom, Core i7 and others.

6.1. High Performance DSP Applications

Digital signal processors process analog signals but high performance digital signal processing systems process digital signals rapidly. One of high performance applications of DSP is wireless base station. System designers of the high performance DSP were given limited option of choice but some of manufactures produce series of programmable DSP parts. DSP processors by Intel, Analog Devices [59], Motorolla, Texas Instrument improved already existed architecture by improving performance achieved by improving clock speed and reducing power consumption [17].

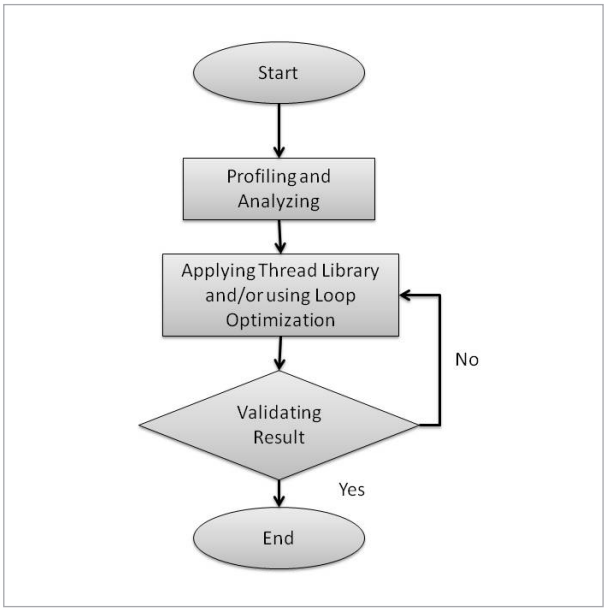
Many high performance applications, for example, data analytics applications, image processing and/or graphic processing applications required machine learning techniques on big data which required extensive computational power. Faella [19] compared DSP processor, GPU and Intel Core i7 to analyze the performance of these processors for support vector machine (SVM) algorithm. It was observed that last level cache was better utilized and needed for the implementation of the proposed method. The performance measure done by the author shows lower clock speed for DSP processors than Intel i7 processor. A performance comparison of GPUs with Intel i7 showed that although GPU clock speed was slower than Intel i7, it showed better performance with performance improvement techniques. This may be attributed either due to its adaptability of different computational intensive algorithms or its architectural support for parallel computations.

6.2. Bioinformatics Applications

Bioinformatics applications required higher performance computation for analyzing huge amount of data. Current methodologies and computing power are not sufficient enough to surpass this bottleneck [22]. However, many manufacturers provide multicore processors and/or prototypes to achieve the high performance computation like Intel Terascale Processors [45] or Microsystems UltraSPARC T2 [64].

Intel Terascale processors provide first level cache L1 to each core and numerous levels of L2 cache. However, researches are carried on to explore the possibility of large and low latency last level cache using 3D-stacking [29].

Figure 4
Two level strategies on multicore processor [11]



Still many applications required some parallelization algorithms since they are not able to fully exploit the power provided by these processors. Galvez et al. [22] provided such parallelism algorithm (named as Fast LSA) in order to fully exploit the computational power. The algorithm was tested on different benchmarks, which shows great performance on a standalone general-purpose multicore chip. The performance of an algorithm that did not require floating point calculations was measured on Multicore64-NW, and it obtained 20 times faster optimal alignment. The algorithm exploits the three level cache of TilExpress-20G cards along with improved MESH for communication between cores, cache and shared memory. Another optimizing technique utilizing caches was proposed by Chaichoompu et al. [11]. The authors proposed multithreading and vectorizing strategies to improve performance. To exploit multithreading in bioinformatics applications, the authors proposed a compiler optimization strategy to perform software profiling in order to analyze and distinguish the portions which bioinformatics tool can improve or those which cannot be improved. Later those code parts, which are not executed in a sequential manner, were modified by thread library or via loop optimization technique. The optimized code was verified in a final step to compare the results on multiple cores processors. Intel Core 2 Duo, Intel Core Solo, Intel Pentium 4 were used in order to test the proposed algorithm. These all processors own two level of caches in which last level cache is L2. Figure 4 shows the flow of the algorithm.

6.3. IOT Based Applications

In an Internet era, high computing devices are growing and evolving Internet of things (IoT). Intelligence is embedded into devices which are network of sensors, actuators or processors to ease daily life style.

From short range transceivers to high impact gadgets [3], all sorts of devices ranging from those of automotive industry to those of aerospace, from infrastructure to medical services, from defense industry to daily house hold items, and in many more other areas, IoT-based applications have so much influence to improve our way of life. According to statistics, 70 billion of devices will be connected to Internet in the 2020. However, the discussed field is still growing and researchers are working on optimizing the already existed solutions and finding solutions for complex problems.

For high computation, many IoT devices used GPU or Hybrid GPU/CPU approaches. GPUs are multicore architecture which are highly parallel and use multithreading [20, 32]. Modern GPUs, including GigaByte GTX and NVIDIA Geforce, have hundreds of processing units which achieve massive arithmetic calculation [38]. For high-performance computing, there are high-performance optimized GPUs that help to compute big data in data centers. However, their performance requirement based on sufficient parallelism, combine memory access and/or coherent execution among threads. Many applications have non regular memory access pattern and GPU caches exhibit poor performance if there is mismatch in cache hierarchy design [14]. In GPUs, memory latency is usually not hidden in larger caches, hence GPUs use multithreading to hide latency, but it is useful only for applications which use multithreading.

To avoid poor performance and to achieve easy communication between threads, some GPU manufacturers improved memory hierarchy design and threading communication. NVIDIA introduced Compute Unified Device Architecture (CUDA), which provides parallel computing platform and runs on hundreds of GPU processor core and is highly parallel in nature [33]. Due to intensive computational power, it is much faster than CPU and is used in many high-performance computing applications. The installed shared memory has low-latency and it is plugged-in near each processor core [55]. CUDA based GPUs are introduced with global and shared memory access, where global memory accesses are always cached in L2 cache.

6.4. Image and Video Processing Applications

Many high computational applications require high performance processors to carry out extensive computation in less time and given memory storage. Sometimes it is hard to choose between different processors, which can cater application needs. Evaluating the performance of processor before choosing one, is the key to satisfy application computational needs. Many authors try to evaluate processor's performance on different basis, like time to perform one single operation, memory consumption or energy consumption of processor and have proposed different techniques to optimize the performance by optimizing cache or last level cache. Asaduzzaman and Mahgoub [2] eval-

uated the performance of cache in DSP processors for MPEG4 applications using different cache sizes. They proposed simulation program which optimizes the cache size for task rate. The authors have run a simulation program against 384KB, 512KB and 1024KB. For 384KB cache size, proposed simulation fails since DSP utilization reached beyond 100% usage but, for 512KB and 1024 cache size, performance improved but it did not impact on DSP utilization. Hoozemans et. al. [31] evaluated the performance of VLIW processors, rVEX and Xilinx MicroBlaze for high resolution image processing applications. The authors conclude that rVEX processors gave 80% faster result for image processing convolution algorithm and 2.3 to 3 factor times better result for grayscale conversion. Blair et al. [4] evaluated by comparing FPGA, GPU and DSP performance for image processing and computer vision algorithms. The authors compare performance on different algorithms for the execution time to process images or computer visions. It has been stated that GPU performance is better than FPGA implementation on DSP slices.

7. Challenges

This section discusses the challenges and limitations of data replication and cache partitioning techniques in Last Level Cache.

7.1. Data Replication

Data replication is one of the most effective and highly researched technique, used to handle memory latency issue, since it increases the availability of data, and enhances the performance and reliability [23]. However, it comes with overheads too, e.g., if the environment deals only with read-only request, then performance will be increased significantly (as discussed and proved in Section 3), otherwise for write requests there is a need to maintain consistency among replicas.

7.1.1. Storage

Replication techniques come with storage overhead which cannot be dealt with beyond some extent, since replication techniques do not only rely on creating multiple copies, but they also need to maintain the consistency among them. Some of the techniques [39, 70] use additional classifier or tags, which also requires additional capacity to store information about replicated data. Kurian et al. [39] showed that locality classifier, which is required to maintain consistency, took additional 30% of storage overhead for 64 cores and it increase gradually as the number of cores are added in the system. Classifier used to maintain consistency took extra 36 bits for each entry in the 64-core processor which ultimately costs 60% more storage than the baseline protocol as discussed in [39].

Storage overhead for tag replication is massive [70], due to this reason the researchers prefer to use selective replication approach. Nevertheless, storage overhead still manages to reach 15.6%, which ultimately may cause performance degradation, considering a gradual increase in data replication and failures in the network.

7.1.2. Maintaining Consistency

All data replication techniques come in packaged with the protocols and methodologies for maintaining consistency. These protocols and methodologies covers consistency issues, however, achieving full consistency is impossible and some inconsistencies are still present in the system.

There is a chance of conflict, when updating replicas, e.g., if the number of updates increases (due to more replicas), then there is a possibility of conflict among updates at the same time. If the number of replicas are increased, then there is a chance that consistency becomes weak due to difficulty in maintaining consistency among all replicas (e.g., insufficient network bandwidth).

7.1.3. Network Traffic

All replications involving consistency models and approaches are the reason for additional network traffic and bandwidth requirement [47]. To maintain consistency, there is a need to update every single data item replicated, which in turn results in additional operation of write-update requests, including original data access or write request. These protocols need extra acknowledgment transactions in case of data invalidation or eviction [24, 40, 70].

7.1.4. Additional Power Consumption

Data replication techniques are proposed to minimize memory latency and increase energy efficiency, however, protocols incurred with consistency are the rea-

son for additional energy consumption of system. For example, if replicas are greater in number and there is a need to update or fetch replicated data and processor is busy with some other task, then request will be piled up and ultimately all bandwidth will be used to fulfill an on ongoing operation. This situation ultimately increases energy consumption of the system.

7.2. Cache Partitioning

Cache partitioning is only useful when cache size is not large [50]. It is harmful in some extent for the applications exploiting locality [12], however, it is useful for the application which relies on last level cache [52]. There are other similar limitations as well.

7.2.1. LRU Replacement

LRU is considered to be the one of the best replacement techniques in case of data eviction, and designers opt the policy in architecture to identify the data need to be replaced. However, studies have discussed [48, 49, 50] and some researches proved that if LRU policies are not optimized according to competing resources, it can degrade performance [54]. For example, some of LRU based replacement policies do not work with some of cache partitioning techniques which are designed considering full-associative caches [52] and applied to set-associate caches [52].

7.2.2. Performance Overhead

Despite the advantage of cache partitioning techniques there is still performance overhead due to load-imbalance, handling of different miss latency via same approach, bandwidth congestion due to network traffic etc. Some of cache partitioning techniques try to overcome these overheads, for example, miss rate penalty, but still they lack to fully cover these issues. Limitation of LRU policy as discussed in [69] is also one of the reasons for performance overhead.

7.2.3. Design Choice

Cache partitioning techniques are used not only to optimize performance of cache but to achieve Quality of Service (QoS), improving energy efficiency or load balancing etc. To achieve these goals, one needs to carefully select parameter for cache partitioning techniques on different architecture. Cache partitioning techniques include parameters like replacement policies, quota allocation, partitioning interval, etc. [53].

8. Conclusion

Technology is advancing at break-neck speed and with it the requirements in terms of application processing time and manipulation of real-time data. This research article explores the different dynamics in improving overall last level cache performance. It particularly focused on various optimizing techniques such as *Data Replication* and *Cache Partitioning*. Energy efficiency is an important design issue in multicore processors, hence the article have comprehensively compared different mechanisms addressing energy saving. The manuscript surveyed different authors' work which evaluates processors performance in high performance applications. Finally, there are always challenges and trade-offs in implementing any optimizing technique, which have been discussed in detail.

References

1. Abella, J., González, A., Vera, X., O'Boyle, M. F. IATAC: A Smart Predictor to Turn-Off L2 Cache Lines. ACM Transactions on Architecture and Code Optimization (TACO), 2005, 2(1), 55-77. https://doi.org/10.1145/1061267.1061271
2. Asaduzzaman, A., Mahgoub, I. Evaluation of Application-Specific Multiprocessor Mobile System. Proceedings of the 2004 Symposium on Performance Evaluation of Computer Telecommunication Systems, 2004, 751-758.
3. Bandyopadhyay, D., Sen, J. Internet of Things: Applications and Challenges in Technology and Standardization. Wireless Personal Communications, 2011, 58(1), 49-69. https://doi.org/10.1007/s11277-011-0288-5
4. Blair, E., Redwood, C., Ashrafian, H., Oliveira, M., Broxholme, J., Kerr, B., Salmon, A., Östman-Smith, I., Watkins, H. Mutations in the 2 Subunit of AMP-Activated Protein Kinase Cause Familial Hypertrophic Cardiomyopathy: Evidence for the Central Role of Energy Compromise in Disease Pathogenesis. Human Molec-

ular Genetics, 2001, 10(11), 1215-1220. https://doi.org/10.1093/hmg/10.11.1215

5. Bosse, T., Hoogendoorn, M., Memon, Z. A., Treur, J., Umair, M. An Adaptive Model for Dynamics of Desiring and Feeling Based on Hebbian Learning. In International Conference on Brain Informatics, Springer, 2010, 14-28. https://doi.org/10.1007/978-3-642-15314-3_3
6. Bosse, T., Hoogendoorn, M., Memon, Z. A., Treur, J., Umair, M. A Computational Model for Dynamics of Desiring and Feeling. Cognitive Systems Research, 2012, 19-20, 39-61. https://doi.org/10.1016/j.cogsys.2012.04.002
7. Bosse, T., Memon, Z. A., Treur, J., Umair, M. An Adaptive Human-Aware Software Agent Supporting Attention-Demanding Tasks. International Conference on Principles and Practice of Multi-Agent Systems, Springer, 2009, 292-307. https://doi.org/10.1007/978-3-642-11161-7_20
8. Bosse, T., Memon, Z. A., Treur, J. Emergent Storylines Based on Autonomous Characters with Mindreading Capabilities. IEEE/WIC/ACM International Conference on Intelligent Agent Technology, 2007 (IAT'07), 2007, 207-214. https://doi.org/10.1109/IAT.2007.61
9. Bosse, T., Memon, Z. A., Treur, J. Modelling Animal Behaviour Based on Interpretation of Another Animal's Behaviour. ICCM'07: Proceedings of the 8th International Conference on Cognitive Modeling, 2007, 193-198.
10. Bosse, T., Memon, Z. A., Treur, J. Adaptive Estimation of Emotion Generation for an Ambient Agent Model. European Conference on Ambient Intelligence, Springer, 2008, 141-156. https://doi.org/10.1007/978-3-540-89617-3_10
11. Chaichoompu, K., Kittitornkun, S., Tongsima, S. Speed-up Bioinformatics Applications on Multicore-Based Processor Using Vectorizing and Multithreading Strategies. Bioinformation, 2007, 2(5), 182-184. https://doi.org/10.6026/97320630002182
12. Chang, J., Sohi, G. S. Cooperative Cache Partitioning for Chip Multiprocessors. ACM International Conference on Supercomputing, 25th Anniversary Volume, 2014, 402-412. https://doi.org/10.1145/2591635.2667188
13. Chen, G., Huang, K., Knoll, A. Energy Optimization for Real-Time Multiprocessor System-on-Chip with Optimal DVFS and DPM Combination. ACM Transactions on Embedded Computing Systems (TECS), 2014, 13(3S), 111:1-111:21.
14. Chen, X., Chang, L.-W., Rodrigues, C. I., Lv, J., Wang, Z., Hwu, W.-M. Adaptive Cache Management for Energy-Efficient GPU Computing. 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, 2014, 343-355. https://doi.org/10.1109/MICRO.2014.11
15. Chishti, Z., Powell, M. D., Vijaykumar, T. Optimizing Replication, Communication, and Capacity Allocation in CMPs. 32nd International Symposium on Computer Architecture (ISCA'05), Madison, WI, USA, 2005, 357-368. https://doi.org/10.1109/ISCA.2005.39
16. Cook, H., Moreto, M., Bird, S., Dao, K., Patterson, D. A., Asanovic, K. A Hardware Evaluation of Cache Partitioning to Improve Utilization and Energy-Efficiency While Preserving Responsiveness. ACM SIGARCH Computer Architecture News, 2013, 41(3), 308-319. https://doi.org/10.1145/2508148.2485949
17. Curtis, T., Curtis, M. High Performance Digital Signal Processing. IOA Conference on Sonar Signal Processing, 2004, 1-10.
18. Do, C. T., Kim, J., Hwang, I., Kim, S.-H., Kim, C. H. A Novel Last-Level Cache Replacement Policy to Improve the Performance of Mobile Systems. Advanced Science and Technology Letters (Workshop on Mobile and Wireless 2014), 2014, 46. https://doi.org/10.14257/astl.2014.46.06
19. Faella, J. On Performance of GPU and DSP Architectures for Computationally Intensive Applications, 2013.
20. Fernando, R. GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics. Pearson Higher Education, 2004.
21. Flautner, K., Kim, N. S., Martin, S., Blaauw, D., Mudge, T. Drowsy Caches: Simple Techniques for Reducing Leakage Power. IEEE Proceedings of the 29th Annual International Symposium on Computer Architecture, 2002, 148-157.
22. Gálvez, S., Daz, D., Hernández, P., Esteban, F. J., Caballero, J. A., Dorado, G. Next-Generation Bioinformatics: Using Many-Core Processor Architecture to Develop a Web Service for Sequence Alignment. Bioinformatics, 2010, 26(5), 683-686. https://doi.org/10.1093/bioinformatics/btq017
23. Goel, S., Buyya, R. Data Replication Strategies in Wide-Area Distributed Systems. Enterprise Service Computing: From Concept to Deployment, IGI Global, 2007, 211-241. https://doi.org/10.4018/978-1-59904-180-3.ch009
24. Gracioli, G., Fröhlich, A. A., Pellizzoni, R., Fischmeister, S. Implementation and Evaluation of Global and Partitioned Scheduling in a Real-Time OS. Real-Time

Systems, 2013, 49(6), 669-714. https://doi.org/10.1007/s11241-013-9183-3

25. Gracioli, G., Frohlich, A. A. An Experimental Evaluation of the Cache Partitioning Impact on Multicore Real-Time Schedulers. 2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2013, 72-81.
26. Gupta, S., Zhou, H. Spatial Locality-Aware Cache Partitioning for Effective Cache Sharing. IEEE 2015 44th International Conference on Parallel Processing (ICPP), 2015, 150-159. https://doi.org/10.1109/ICPP.2015.24
27. Hameed, H., Durrani, N. M., Hina, S., Shamsi, J. A. Towards Efficient Graph Traversal Using a Multi-GPU Cluster. International Journal of Advanced Computer Science and Applications, 2017, 8(6), 338-346. https://doi.org/10.14569/IJACSA.2017.080644
28. Hardavellas, N., Ferdman, M., Falsafi, B., Ailamaki, A. Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches. ACM SIGARCH Computer Architecture News, 2009, 37(3), 184-195. https://doi.org/10.1145/1555815.1555779
29. Held, J., Bautista, J., Koehl, S. From a Few Cores to Many: A Tera-Scale Computing Research Overview. White Paper, Intel, 2006.
30. Hoogendoorn, M., Klein, M. C., Memon, Z. A., Treur, J. Formal Specification and Analysis of Intelligent Agents for Model-Based Medicine Usage Management. Computers in Biology and Medicine, 2013, 43(5), 444-457. https://doi.org/10.1016/j.compbiomed.2013.01.021
31. Hoozemans, J., Wong, S., Al-Ars, Z. Using Vliw Softcore Processors for Image Processing Applications. IEEE 2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015, 315-318. https://doi.org/10.1109/SAMOS.2015.7363691
32. Hwang, K., Dongarra, J., Fox, G. C. Distributed and Cloud Computing: From Parallel Processing to the Internet of Things, Morgan Kaufmann, 2013.
33. Inam, R. An Introduction to GPGPU Programming-Cuda Architecture, 2010.
34. Kaxiras, S., Hu, Z., Martonosi, M. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. ACM SIGARCH Computer Architecture News, 2001, 29(2), 240-251. https://doi.org/10.1145/384285.379268
35. Khan, F. A., Tahir, M. A., Khelifi, F., Bouridane, A., Almotaeryi, R. Robust Off-Line Text Independent Writer Identification Using Bagged Discrete Cosine Transform Features. Expert Systems with Applications, 2017, 71, 404-415. https://doi.org/10.1016/j.eswa.2016.11.012
36. Khan, M. A., Memon, Z. A., Khan, S. Highly Available Hadoop Namenode Architecture. In IEEE 2012 International Conference on Advanced Computer Science Applications and Technologies (ACSAT), 2012, 167-172. https://doi.org/10.1109/ACSAT.2012.52
37. Kim, C., Burger, D., Keckler, S. W. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. ACM Sigplan Notices, 2002, 37(10), 211-222. https://doi.org/10.1145/605432.605420
38. Kindratenko, V. V., Enos, J. J., Shi, G., Showerman, M. T., Arnold, G. W., Stone, J. E., Phillips, J. C., Hwu, W.-M. GPU Clusters for High-Performance Computing. IEEE International Conference on Cluster Computing and Workshops, CLUSTER'09, 2009, 1-8. https://doi.org/10.1109/CLUSTR.2009.5289128
39. Kurian, G., Devadas, S., Khan, O. Locality-Aware Data Replication in the Last-Level Cache. 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), 2014, 1-12.
40. Kurian, G., Khan, O., Devadas, S. The Locality-Aware Adaptive Cache Coherence Protocol. ACM SIGARCH Computer Architecture News, 2013, 41(3), 523-534. https://doi.org/10.1145/2508148.2485967
41. Li, L., Kadayif, I., Tsai, Y.-F., Vijaykrishnan, N., Kandemir, M., Irwin, M. J., Sivasubramaniam, A. Managing Leakage Energy in Cache Hierarchies. Journal of Instruction-Level Parallelism, 2003, 5, 1-24.
42. Li, Y., Skadron, K., Lee, B., Brooks, D. Quantifying Latency and Throughput Compromises in CMP Design. Technical Report CS-2006-26, University of Virginia Department of Computer Science, Technical Report, 2006.
43. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., Hazelwood, K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. ACM Sigplan Notices, 2005, 40(6), 190-200. https://doi.org/10.1145/1064978.1065034
44. Lu, J., Guo, Y. Energy-Aware Fixed-Priority Multi-Core Scheduling for Real-Time Systems. 2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2011, 1, 277-281.
45. Mattson, T. G., Van der Wijngaart, R., Frumkin, M. Programming the Intel 80-Core Network-on-a-Chip Terascale Processor. Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, 2008, 1-11. https://doi.org/10.1109/SC.2008.5213921

46. Memon, Z. A., Treur, J. Modelling the Reciprocal Interaction Between Believing and Feeling from a Neurological Perspective. International Conference on Brain Informatics, Springer, 2009, 13-24. https://doi.org/10.1007/978-3-642-04954-5_12

47. Memon, Z. A., Treur, J. On the Reciprocal Interaction Between Believing and Feeling: An Adaptive Agent Modelling Perspective. Cognitive Neurodynamics, 2010, 4(4), 377-394. https://doi.org/10.1007/s11571-010-9136-7

48. Memon, Z. A., Treur, J. An Agent Model for Cognitive and Affective Empathic Understanding of Other Agents. Transactions on Computational Collective Intelligence VI, Springer, 2012, 56-83.

49. Memon, Z., Treur, J. Cognitive and Biological Agent Models for Emotion Reading. 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, 2008, 308-313. https://doi.org/10.1109/WIIAT.2008.311

50. Mittal, S., Vetter, J. S. A Survey of Techniques for Architecting DRAM Caches. IEEE Transactions on Parallel and Distributed Systems, 2016, 27(6), 1852-1863. https://doi.org/10.1109/TPDS.2015.2461155

51. Mittal, S., Zhang, Z. Palette: A Cache Leakage Energy Saving Technique for Green Computing. Transition of HPC Towards Exascale Computing, 2013, 24, 46-61.

52. Mittal, S. A Survey of Techniques for Cache Partitioning in Multicore Processors. ACM Computing Surveys (CSUR), 2017, 50(2), 27:1-27:39.

53. Moreto, M., Cazorla, F. J., Ramirez, A., Sakellariou, R., Valero, M. FlexDCP: A QoS Framework for CMP Architectures. ACM SIGOPS Operating Systems Review, 2009, 43(2), 86-96. https://doi.org/10.1145/1531793.1531806

54. Nikas, K. An Analysis of Cache Partitioning Techniques for Chip Multiprocessor Systems. Ph.D. Dissertation, University of Manchester, 2008.

55. Nvidia, C. Nvidia Cuda C Programming Guide. Nvidia Corporation, 2011, 120(18), 8.

56. Olukotun, K., Hammond, L., Laudon, J. Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency. Synthesis Lectures on Computer Architecture, 2007, 2(1), 1-145. https://doi.org/10.2200/S00093ED1V01Y200707CAC003

57. Powell, M., Yang, S.-H., Falsafi, B., Roy, K., Vijaykumar, T. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. ISLPED'00: Proceedings of the 2000 International Symposium on Low Power Electronics and Design (Cat. No.00TH8514), Rapallo, Italy, 2000, 90-95. https://doi.org/10.1109/LPE.2000.155259

58. Qureshi, M. K., Patt, Y. N. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. 39th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-39, 2006, 423-432. https://doi.org/10.1109/MICRO.2006.49

59. ROM, B. Adsp-bf531/adsp-bf532/adsp-bf533.

60. Sanchez, D., Kozyrakis, C. The Zcache: Decoupling Ways and Associativity. 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2010, 187-198. https://doi.org/10.1109/MICRO.2010.20

61. Sanchez, D., Kozyrakis, C. Vantage: Scalable and Efficient Fine-Grain Cache Partitioning. ACM SIGARCH Computer Architecture News, 2011, 39(3), 57-68. https://doi.org/10.1145/2024723.2000073

62. Savera, A., Zia, A., Edhi, M. S., Tauseen, M., Shamsi, J. A. Bumpster: A Mobile Cloud Computing System for Speed Breakers and Ditches. IEEE 41st Conference on Local Computer Networks Workshops (LCN Workshops), 2016, 65-71. https://doi.org/10.1109/LCN.2016.030

63. Seznec, A. A Case for Two-Way Skewed-Associative Caches. In ACM SIGARCH Computer Architecture News, 1993, 21(2), 169-178. https://doi.org/10.1145/173682.165152

64. Shah, M., Barreh, J., Brooks, J., Golla, R., Grohoski, G., Gura, N., Hetherington, R., Jordan, P., Luttrell, M., Olson, C., Saha, B., Sheahan, D., Spracklen, L., Wynn, A. Ultrasparc T2: A Highly-Treaded, Power-Efficient, SPARC SOC. 2007 IEEE Asian Solid-State Circuits Conference, Jeju, 2007, 22-25. https://doi.org/10.1109/ASSCC.2007.4425786

65. Shaikh, M. K., Lawgaly, A., Tahir, M. A., Bouridane, A. Modality Identification for Heterogeneous Face Recognition. Multimedia Tools and Applications, 2017, 76(3), 4635-4650. https://doi.org/10.1007/s11042-016-3635-4

66. Shamsi, J. A. A Laboratory Based Course on GPU Programming: Methods, Practices, and Lessons. Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International, 2017, 367-374. https://doi.org/10.1109/IPDPSW.2017.47

67. Shamsi, J., Brockmeyer, M. Predictable Service Overlay Networks: Predictability Through Adaptive Monitoring and Efficient Overlay Construction and Management. Journal of Parallel and Distributed Computing, 2012, 72(1), 70-82. https://doi.org/10.1016/j.jpdc.2011.09.005

68. Shi, X., Su, F., Peir, J.-k., Xia, Y., Yang, Z. CMP Cache Performance Projection: Accessibility vs. Capacity. ACM