International Journal of Communication and Information Technology

E-ISSN: 2707-6628
P-ISSN: 2707-661X
www.computersciencejournals.com/ijcit
IJCIT 2022; 3(2): 31-35
Received: 05-07-2022
Accepted: 14-08-2022

Manideep Yenugula
Kroger, Blue ash, Ohio, 45242, United State of America

Corresponding Author:
Manideep Yenugula
Kroger, Blue ash, Ohio, 45242, United State of America

Examining partitioned caches performance in heterogeneous multi-core processors

Manideep Yenugula

DOI: https://doi.org/10.33545/2707661X.2022.v3.i2a.70

Abstract
The last-level cache (LLC) is shared by many distinct kinds of cores in asymmetric multi-core systems. There is greater rivalry in the LLC since different core types have different memory access needs. Our new technique for replacing the split cache, HAPC, takes heterogeneity into account. To improve core-to-core interference, this method uses cache partitioning. In multithreaded applications, it guides the replacement strategy by monitoring the shared reuse state of every cache block inside the partition during runtime. This ensures that cache blocks shared among different cores are maintained. For huge cores to make more efficient LLC visits, cache replacement algorithms generally modify the leftover state while preserving cache blocks needed by them. This approach takes into consideration the fact that heterogeneous cores have different memory accesses to LLC. In comparison to the state-of-the-art cache substitute's algorithms, LRU and SRCP, HAPC can greatly enhance the performance of large cores when running multithreaded applications, with almost no impact on small cores, leading to an overall improvement in system performance.

Keywords: Asymmetric multi-core processors, L2 cache, last level cache, cache replacement policy, CPU power

Introduction
Due to task interference in the shared caches, timing verification of real-time software activities becomes very challenging on multi-core CPUs. When it comes to multi-core processors having shared caches, two main ways to improve scheduling efficiency are cache partitioning and explicitly measuring the quantity of cache interference across activities. The first method has poor schedulability performance because of very negative cache interference estimates, whereas the second method may cause jobs to take longer to execute because of less cache utilisation, which is bad for schedulability as well [1]. As the lag time among processing power with main memory access delay has increased, cache memory has become an integral part of CPU design. Through a multi-tiered cache architecture, processors support anything from the tiniest and quickest private caches (Level 1) to the biggest and slowest caches (Level 2). To guarantee the consistency of private cache data, multi-core processors now incorporate Snoop Control Unit [2]. Modern real-time embedded systems are drawn to multi-core processors due to their reduced power consumption, smaller size, and faster performance. Meeting these strict deadlines while maintaining system accuracy and predictability and attaining improved performance is of the utmost importance in real-time computing. Tasks' worst-case execution times rise when numerous processes using different cores access shared resources (such as memory, direct memory access, and peripherals) at the same time, which increases blocking time and increases the likelihood that tasks will miss their hard deadlines, which can lead to system failure [3]. Because of their criticality, avionics systems are heavily controlled and need determinism. Because there is just one processor in a single-core system, there is only one possible execution flow, which makes the system very predictable. Nevertheless, production of multi-core CPUs is becoming the only focus of manufacturers. The use of multi-core architectures in avionics systems is therefore mandated. To take use of the parallelism given by multi-core architectures, mechanisms to alleviate the loss of predictability must be created [4]. By consolidating several operations into a single chip, multi-core processors are anticipated to enhance performance and decrease production costs in embedded systems like automotive systems. However, deadline miss rates for such systems are raised due to inter-core interfering in shared last-level cache, which causes time-sensitive tasks with (soft) timing restrictions to have unpredictable and

increased execution delays [5]. A shared last levels cache is a feature of most current multi-core processors that allows data from all cores to be stored, which improves speed. Nevertheless, cache pollution presents a fresh obstacle for cache management. When two sets of mapped data have different degrees of temporal locality, cache pollution occurs and data with weaker temporal locality is pushed out of the cache set [6]. The current development in vehicle systems is to reduce cost and boost efficiency by integrating more software programmes into fewer ECUs. Congestion on resources like caches might occur as a result of additional apps using the same resources. Applications that rely on time-sensitive data may encounter problems using shared resource congestion due to the unpredictable nature of application interference. Implementing cache partitioning methods is one potential solution to the issue of shared resource overload. These strategies divide up the available cache lines across different applications [7].

Our proposal is a framework for heterogeneous-aware partitioning cache replacement policies (HAPC), which will lead to the AMP system operating at peak efficiency. To keep core interference to a minimum, HAPC takes cache block reuse into account while replacing cache. The main concept of HAPC is to detect the LLC needs of heterogeneous cores and direct the replacement strategy based on runtime analysis of core and thread memory access patterns. The following are some of the overarching contributions made by this paper.

- We propose a heterogeneous-aware partitioning cached replacement strategy to safeguard against inter core interference and improve data usage effectiveness in partitioning LLC in an asymmetrical multi-core architecture.
- A usage count table (RCT) is kept with the historical reuse data for every LLC cache block; this information is helpful for making decisions about when to replace caches. We next adjust the number of cores in the RCT according on the sequences of access to memory for both large and tiny cores.
- Last but not least, we thoroughly evaluate HAPC using PARSEC 3.0 using the gem5 simulator. Compared to LRU and SRCP, HAPC often boosts large core performance by 4.57% and 2.44%, respectively. Compared to conventional replacement policies, HPAC may significantly boost performance across a range of workloads and system setups.

The paper's second section provides the necessary context and relevant literature. In Section 3, our HAPC structure is detailed. The experimental procedure and data analysis are covered in the fourth section. Section 5 wraps up the paper.

Related Work

Partitioned cache replacement requirements that take heterogeneity into account are introduced in the present study by the authors of [8]. This strategy uses runtime tracking of the shared reuse condition of each cache block within the partition to guide the replacement process, ensuring that cache blocks utilized by various cores in multithreaded applications are kept. This is accomplished by using cache partitioning, which effectively decreases the mutual interference across cores. To boost the efficiency of huge cores' LLC accesses, the cache replacement strategy often keeps cache blocks needed by them while changing the reuse state, taking into account the disparity in memory accesses to LLC via heterogeneous cores. When multithreaded programs are executed, HAPC may boost the system's speed by making large cores work better while tiny cores are affected. This is in contrast to two cutting-edge cache replacement algorithms, LRU and the SRCP.

To address the needs of non-preemptive multi-core systems operating in real-time with partitioned caches, TCPS, a heuristic partitioned planner, was proposed in [9]. By using task characteristics and combining the advantages of partitioned scheduling and cache partitioning, TCPS achieves an elevated level of schedulability while alleviating computing power from contention and minimising cache interference, respectively. To test TCPS's schedulability and see how it stacked up against other global as well as partitioned scheduling methods, batteries of thorough tests were run. In regard to schedulability, their findings demonstrate that TCPS is the most effective scheduling strategy, leading to more consistent load balancing and better cache utilisation.

Partitioned scheduling with shared cache interference is another typical scheduling paradigm considered in [10]. Researchers provide CITTA, a task partitioning technique that takes cache interference into account, to accomplish this. They begin by dissecting the shared cache disturbance that occurs when two data cache and set-associative instruction programmes work together. A task's upper limit on cache interference is then computed using an integer programming formulation, as specified by CITTA. They officially demonstrate the correctness of CITTA by a schedulability study. On both real and simulated embedded system workloads, many tests are conducted to assess CITTA's schedulability in comparison to other greedy partition methods like First-fit and Worst-fit, as well as global EDF scheduling. Based on their empirical assessments, CITTA performs better than global EDF planning and greedy partition approaches on schedulable task sets.

The existing literature on the topic of application-specific cache alignment algorithms is covered in [11]. For real-time operating systems that support symmetric or asymmetric multi-processing, there will be a noticeable increase in system overhead if inter-core communication is implemented at the OS level. This work presents the first of its kind a novel core-to-core communications approach that relies on cache-aware data lookups. The Little Kernel organizer integrates this capability as "SCHEDULE-ASSIST" and "ENTITY-ASSIST" in the SMP RTOS scheduler. They show, via analytical modeling as well as execution utilizing a Xilinx Evaluating Kit, that RTOS processors Key Performing Indices for RTOS's APIs may be up to 13% more efficient than techniques that don't take the cache architecture into account.

Another popular scheduling paradigm is examined in the study [12]. Partitioned scheduling with shared cache interference. Their proposed cache-interference-aware task partitioning technique, CITTA, is designed to do this. In order to determine the highest limit on cache interference that a job may display, as is necessary for CITTA, an integer programming approach is developed. Authors officially

demonstrate the correctness of CITTA by a schedulability study. They do some tests to see how well CITTA fares in terms of schedulability when compared to global EDF scheduling on task sets that are completely at random. Regarding the sets of tasks that are considered schedulable, their empirical tests reveal that CITTA performs better than global EDF scheduling.

By grouping tasks according to their constraints, dependencies, as well as preferences, and then allocating these groups over several cores, the authors of [13] suggest an Inter-task Affinity-aware Tasks Assignment (IATA) method that minimises the additive overheads in WCET. Dedicated I/O cores handle sluggish I/O peripherals, while scratch-pad RAM is used for high-priority activities that share datasets in order to decrease cache evictions. Their new technique, IATA, is evaluated by comparing it to the blocking-agnostic Best Fit Reducing (BFD) algorithm for task allocation. Their suggested IATA method improves schedulability, decreases CPU utilisation, and assigns an average of 97.5% greater task-sets than BFD, according to the findings.

The emphasis in [14] is on avionics Real-Time Operating Systems and partitioned systems in general. Authors provide a framework that examines application behaviour using traces. It has an inbuilt cache simulator that provides cache information as well. They used a cache locking choice approach to verify their system. After locking data in the cache, their framework with its embedded simulator conducts tests with improved execution predictability and approximately 25% less interference.

A shared LLC design that takes time sensitivity into account was suggested in [15] to address these issues. To begin, the suggested LLC architecture may be able to recognize TST data and instructions through incorporating a time-sensitivity indicator bit into every cache block. Secondly, a time-sensitivity-aware dead block-based caches partition approach is developed and implemented to allocate some of the LLC space to general operations in a way that does not impact TSTs. Thirdly, researchers put forth a cache partitioning strategy that takes memory access characteristics, time-sensitivity tasks (TATS) and managing tasks in shared caches into account in order to further decrease the time limit miss proportion of TSTs. Incorporating their suggested dead block-based technique with the TATS was their goal. Compare TSTs to traditional shared caches, and our research reveals that the suggested methods significantly cut down on deadline misses. Their suggested dead block-based cache partitioning decreases average partitioning by 30.5%, state-of-the-art quality-of-service-aware segmentation by 2.6%, and average deadline miss rates by 9.3% on a dual-core system. Reduced by 21.2% on a quad-core system, 17.7% on equal partitioning, and 4.1% on state-of-the-art quality-of-service-aware caches segmentation are the baseline average deadline miss rates with our suggested solutions.

Proposed Work

System Model

Based on their access rate in the reuse counting table (RCT), the heterogeneous-aware partitioning cached replacement strategy (HAPC) monitors the reuse status of cached blocks. Blocks having low access rates have their priority for caching lowered, while blocks with high access rates have their priority enhanced, in the LLC. It also increases the number of times the cache block may be reused. Using a block from the lower priority group, the replacement policy chooses to remove a cache block that hasn't been used recently. The RCT may further classify memory access requests by differentiating between those made by the shared core as opposed to those produced by the local core. The LC counters in the RCT record each time the cache block is reused by the local core. The core's cache block data becomes increasingly localized and often accessed as the LC inverse becomes bigger. The SC identification in the RCT identifies and tracks a shared cached block whenever it is reused by the other core. A substantial SC score for the cache unit indicates good shared reuse characteristics.

Both the LC and SC numbers begin at zero in the RCT counter. While the LLC access to memory demand is running, the cache controller modifies the RCT value by following the steps in Algorithms 1. Thus, this change in RCT value guides the technique for replacing cached data. When a cache hit happens, the cache controller adjusts the LC and SC counts to make various reuse weights count, finds out which cores are large and which ones are tiny, and traces the source of the memory access demand. The RCT counter's output determines which cache block should be removed in the event of a cache miss. In the cache partition process, all remaining blocks' LC and SC counter values will be decreased by 1 simultaneously. When the cache block is no longer needed, the replacement method will opt to destroy it, causing its priority to decline. Cache pollution may occur if the block is stored for too long, but this avoids it. The initial value of the LC of each cache block added to the cached partition is the average of all the cache blocks in the partition minus one.

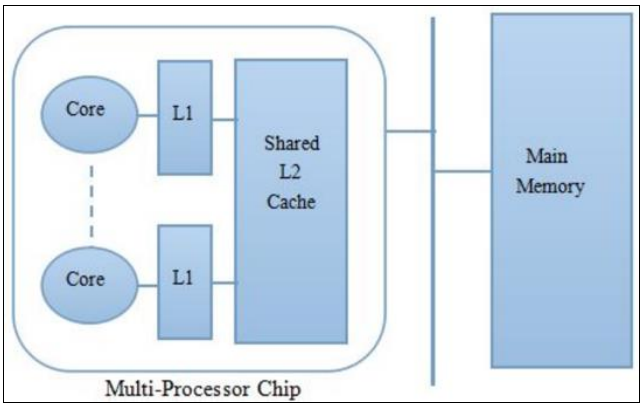

Fig 1: The architecture overview of a multi-core processor

Cache Replacement Policy

It is critical to keep the reuse count table in mind when working with asymmetric multi-core design and heterogeneous-aware partition cache substitution rules, since each core has its own unique memory access behavior traits. During program execution, cache items reused by huge cores will be given more priority by monitoring their performance indicators and dynamically adjusting the RCT reuse proportions (weight big and weight little). The RCT table's reuse count increases in response to high core access and cache blocks that are difficult to delete under a replacement policy; the effect of this is directly proportional to the number of small and large weight relative values. One

measure of large core performance is its memory access success rate. The application's functionality is often tested during runtime. The large core's hit rate for store access is computed and its reuse weight is adjusted at regular intervals. Here is the whole method.

1. Just put 1 in weight big and weight little.
2. The success rate of the large cores in this period should be calculated, and the weight big should be increased by 1 in each subsequent interval until it reaches the threshold, which is the total amount of big cores.
3. Find the large core's hit rate for the next period and compare it to this one. Continue to step 2 if the percentage of hits rises; otherwise, return to step 1.

It is easy to apply the suggested static analysis method to multicore CPUs that have a multi-level memory structure; without sacrificing generalizability, we imagine a dual-core CPU with two layers of cache memories. In a typical dualcore the processor, as shown in Figure 1, each core has its own L1 cache for instructions and data and uses a shared L2 cache. We assume that each core's L1 data cache is flawless (i.e., there are no L1 information cache misses so that instructions accesses to L2 are unaffected by data accesses) since this work concentrates on assessing the inter-thread disruptions caused by instruction streams. However, in our future work, we intend to investigate the worst-case interthread information interferences.

Results & Discussion

To kick off our research, we employed field-programmable gate arrays (FPGAs) to create multi-core designs with varying cache sizes, which we then used to run programs on both homogeneous and heterogeneous multi-core processors. The next step was to execute a scheduler prototype that used offline profiling to allocate threads. With 15 KB of cache, the HMP outperformed a homogeneous multi-core processors with 16 KB of cache in terms of total cache miss rate, allowing for optimal static scheduling.

Table 1: Baseline configuration

Core	Big Core	Little Core
ISA	ARMv8 (64 bit)	ARMv8 (64 bit)
Frequency	2.0 Hz	1.4 Hz
Pipeline	Out-of-order	Out-of-order
Issue width	6	4
Fetch width	16	4
Pipeline stages	Big core	Little core
L1 cache (I & D)	32 KB/2-way	32 KB/2-way
L2 cache	128 KB/2-way	128 KB/2-way
LLC	1 MB–8 MB/16-way	

Conclusion

Research and development efforts will move towards creating low-power, high-performance processors that use an asymmetric multi-core design as the number of computer systems and applications keeps expanding. Here, we lay forth a plan to swap out the asymmetrical multi-core architecture's heterogeneous-aware partition caches (HAPC). In order to prevent the easy elimination of cache units utilized by big cores under replacement policies, HAPC dynamically adjusts the reusing dimensions of cache blocks based on the different memory access behaviors of heterogeneous cores. Consequently, bigger cores will have less interference from smaller ones while accessing memory. Based on the results of the experiments, HAPC is the best replacement policy, outperforming both the conventional LRU policy and the more modern SRCP strategy, which takes sharing and reuse into account. Improved utilization efficacy of LLC and overall system efficiency are achieved by HAPC via asymmetrical multi-core architecture, which makes better use of the large cores' high-performance capabilities without compromising the tiny cores' performance.

Future Work

Dynamic cache partition is another successful approach to managing shared cache in the setting of multi-core processors. We suggested a policy change for static cache partitioning, but it's equally applicable to dynamic partitioning. Our next project will build on top of dynamic partitioning and investigate heterogeneous-aware partitioned cache replacement policies.

References

1. Nayak S, Panda M. Hardware Partitioning Using Parallel Genetic Algorithm to Improve the Performance of Multi-core CPU; c2020.
2. Ghosh SN, Bhargava L, Sahula V. SRCP: sharing and reuse-aware replacement policy for the partitioned cache in multicore systems. Design Automation for Embedded Systems. 2021;25:193-211.
3. Lefoul J. Performance Assessment and Improvement for Cache Predictability in Multi-Core Based Avionic Systems; c2019.
4. Bui PN, Le M, Hoang B, Ngoc NC, Pham HV. Data Partitioning and Asynchronous Processing to Improve the Embedded Software Performance on Multicore Processors. Informatics and Automation; c2022.
5. Konstantinou D, Nicopoulos C, Lee J, Sirakoulis GC, Dimitrakopoulos G. Smart Fork: Partitioned Multicast Allocation and Switching in Network-on-Chip Routers. In: 2020 IEEE International Symposium on Circuits and Systems (ISCAS); c2020. p. 1-5.
6. Park J, Yeom H, Son Y. Page Reusability-Based Cache Partitioning for Multi-Core Systems. IEEE Transactions on Computers. 2020;69:812-818.
7. Danielsson J, Jägemar M, Behnam M, Seceleanu T, Sjödin M. Run-Time Cache-Partition Controller for Multi-Core Systems. In: IECON 2019-45th Annual Conference of the IEEE Industrial Electronics Society; c2019. p. 4509-4515.
8. Fang J, Kong H, Yang H, Xu Y, Cai M. A Heterogeneity-Aware Replacement Policy for the Partitioned Cache on Asymmetric Multi-Core Architectures. Micromachines; c2022. p. 13.
9. Shen Y, Xiao J, Pimentel AD. TCPS: A task and cache-aware partitioned scheduler for hard real-time multi-core systems. In: Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems; c2022.
10. Xiao J, Shen Y, Pimentel AD. Cache Interference-aware Task Partitioning for Non-preemptive Real-time

Multi-core Systems. ACM Transactions on Embedded Computing Systems (TECS). 2022;21:1-28.
11. B SR, Vrind T, I VR. Effective Cache utilization in Multi-core Real Time Operating System. In: 2022 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT); c2022. p. 1-6.
12. Xiao J, Pimentel AD. CITTA: Cache Interference-aware Task Partitioning for Real-time Multi-core Systems. In: The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems; c2020.
13. Akram N, Zhang Y, Ali S, Amjad HM. Efficient Task Allocation for Real-Time Partitioned Scheduling on Multi-Core Systems. In: 2019 16th International Bhurban Conference on Applied Sciences and Technology (IBCAST); c2019. p. 492-499.
14. Lefoul J, Dugo AT, Magalhães FG, Nicolescu G, Assal D, Ulysse N, *et al*. Simulator-Based Framework towards Improved Cache Predictability for Multi-Core Avionic Systems. In: 2020 Spring Simulation Conference (Spring Sim); c2020. p. 1-12.
15. Lee M, Kim S. Time-sensitivity-aware shared cache architecture for multi-core embedded systems. The Journal of Supercomputing; c2019. p. 1-31.