A Project Report On

"Performance Enhancement and Analysis of Shared Last Level Cache in Heterogeneous Multi-core CPU with GPU using MacSim Simulator"

Submitted to
KIIT Deemed to be University

In Partial Fulfillment of the Requirement for the Award of
BACHELOR'S DEGREE IN COMPUTER SCIENCE AND ENGINEERING

BY

MANASH SANGAM	20051731
SHUBHAM JHA	20051757
SANDESH GHIMIRE	20051753

UNDER THE GUIDANCE OF
PROF. DR. BANCHHANDHI DASH

SCHOOL OF COMPUTER ENGINEERING
KALINGA INSTITUTE OF INDUSTRIAL TECHNOLOGY
BHUBANESWAR, ODISHA - 751024

November 2023

KIIT Deemed to be University

School of Computer Engineering Bhubaneswar,
ODISHA 751024

CERTIFICATE

This is certify that the project entitled

"Performance Enhancement and Analysis of Shared Last Level Cache in Heterogeneous Multi-core CPU with GPU using MacSim Simulator"

submitted by

MANASH SANGAM	20051731
SHUBHAM JHA	20051757
SANDESH GHIMIRE	20051753

is a record of bonafide work carried out by them, in the partial fulfillment of the requirement for the award of Degree of Bachelor of Engineering (Computer Science & Engineering OR Information Technology) at KIIT Deemed to be university, Bhubaneswar. This work is done during year 2022-2023, under our guidance.

Date: 22/11/23

Prof. (Dr.) Banchhanidhi Dash

ACKNOWLEDGEMENTS

We express our deepest gratitude to Prof. (Dr.) Banchhanidhi Dash and Prof. (Dr.) Prasant Kumar Pattnaik of Computer Science Department - KIIT for his expert guidance and continuous encouragement to see that this project rights its target since its commencement to its completion. We feel honoured and privileged to have had the opportunity to work under his mentorship, and his inputs will undoubtedly be valuable assets to our future endeavours.

MANASH SANGAM

SHUBHAM JHA

SANDESH GHIMIRE

ABSTRACT

Effectively managing the last-level cache (LLC) in heterogeneous CPU-GPU simulation environments poses distinct challenges, given the varied characteristics of CPU and GPGPU applications. In contrast to traditional memory-intensive CPU benchmarks, GPGPU applications employ a blend of thread-level parallelism (TLP) and caching to mitigate memory latency. To address these challenges, our study introduces two innovative MLP-enhanced cache management policies: MLP-SLCP and MLP-DRRIP. MLP-SLCP utilizes core sampling to dynamically tailor cache allocations, considering the unique behaviours and access patterns of diverse workloads. This approach ensures fair treatment for both CPU and GPGPU applications, accommodating their distinct requirements. On the other hand, MLP-DRRIP addresses challenges related to interference and cache space utilization by dynamically switching between cache insertion policies specifically tailored for GPGPU applications. Our experimental findings underscore the effectiveness of MLP-SLCP and MLP-DRRIP in optimizing cache utilization, highlighting their potential to elevate overall system performance in heterogeneous architectures.

Keywords: Heterogeneous computing, Cache allocation policies, Memory Level Parallelism (MLP), Core sampling techniques, Interference-aware cache management

Table of Contents

List of Figures

List of Tables

INTRODUCTION

1.1 Heterogeneous Multi-core CPU GPU Architecture

1.1.1 ARCHITECTURES

At this section, we delineate the structural frameworks of CPUs and GPUs along with diverse approaches for integrating GPUs within heterogeneous systems. Additionally, we provide a concise overview of the traditional GPU programming model, highlighting distinctions from CPU programming methods.

GPUs are renowned for their elevated computational potency and memory bandwidth, particularly when juxtaposed with CPUs. To illustrate, considering two recent high-performance processors—the NVIDIA Ampere A100 GPU and the AMD EPYC 7702P CPU—the Ampere A100 showcases a 2.8× increase in 32-bit floating point performance and a 7.6× surge in memory bandwidth compared to the EPYC 7702P. Although these performance differentials often motivate the utilization of GPUs in database research for query processing, it is imperative to recognize that reducing GPUs merely to these advantages oversimplifies their multifaceted capabilities.

However, the comparative analysis of GPUs and CPUs should extend beyond raw performance metrics, acknowledging their distinct optimization for varying use cases. Both processor types grapple with the power wall—constraints necessitating manageable power consumption and heat dissipation. Consequently, CPUs and GPUs adopt divergent design trade-offs driven by specific application demands to attain optimal performance within these constraints. This specialized optimization underscores the importance of selecting the most suitable processor contingent on the nature of the computational problem. Subsequently, we delve into the intricate design considerations motivating the architectures of GPUs and CPUs, providing a detailed contrast of various properties of processor between Ampere A100 for reference and the EPYC 7702P.

1.1.2 COMMON CPUS

The central goal for traditional CPUs revolves around optimizing continuous performance, historically driven by Dennard scaling. Dennard scaling, which links transistor size with operating frequency and voltage, has historically led to increased frequency and processing speed as transistors shrink. This, combined with microarchitecture advances extracting Instruction Level Parallelism (ILP) from instruction streams, has yielded a significant 100× performance boost in recent CPUs compared to earlier models. Techniques such as processor pipelines, branch prediction, speculative execution, and out-of-order execution are employed to minimize pipeline stalls and enhance ILP. However, the exploitation of ILP faces limitations imposed by memory system performance, where data references can stall the pipeline due to memory latency and concurrent memory accesses. Despite substantial caches addressing chronological and geographical data access locality, the performance of data-intensive applications is limited by memory access, particularly when faced with mandatory cache misses. Recent trends suggest a slowdown in scalar performance increases since 2003 due to physical limitations on processor frequencies and the non-energy efficiency of ILP-increasing microarchitecture advances. Consequently, manufacturers are redirecting their focus towards throughput improvement by embracing explicit data parallelism, as seen in multi-core CPUs and the adoption of SMT (Simultaneous Multi-Threading) and SIMD (Single Instruction, Multiple Data) instructions, aligning multi-core CPUs more closely with GPUs.

1.1.3 DEDICATED GPUS

The evolution of GPUs from their original design for specialized graphics rendering in 3D games to versatile processors with generic computing capabilities has been driven by the need for a flexible graphics rendering pipeline accommodating a diverse range of 3D games. Primarily optimized for throughput applications, particularly graphics rendering, GPUs possess distinctive features such as significant data parallelism, latency tolerance, and high demands for memory bandwidth. In contrast to CPUs, which rely on Implicit Instruction Level Parallelism (ILP), GPUs leverage explicit data parallelism to keep processing cores active. With numerous simple processing cores, GPUs achieve nearly linear scaling of processing performance with the transistor budget, differing from CPU microarchitectural enhancements that scale proportionally to the square root of the transistor budget. The latency tolerance in GPUs allows for a reduction in processing frequency and an increase in the number of transistors allocated to processing cores within a given power budget. Rather than minimizing the latency of individual data items through caches and microarchitectural advances, GPUs conceal latency by processing other data items, supporting a significant oversubscription of threads. Each streaming multiprocessor (SM) in an Ampere A100 GPU can execute four independent warps simultaneously, managing 64 different warps awaiting execution at each cycle. The GPU's focus on large L1 caches near processing cores, along with a relatively smaller shared last-level cache, emphasizes streaming data access patterns with limited data reuse in graphics workloads. The GPU's memory subsystem is designed for high memory bandwidth, often utilizing three-dimensional stacked memory with a significantly wider memory data bus compared to CPUs. For instance, the Ampere A100 employs ten 512-bit memory controllers, resulting in a bus width of 5120 bits, showcasing a substantial difference in width.

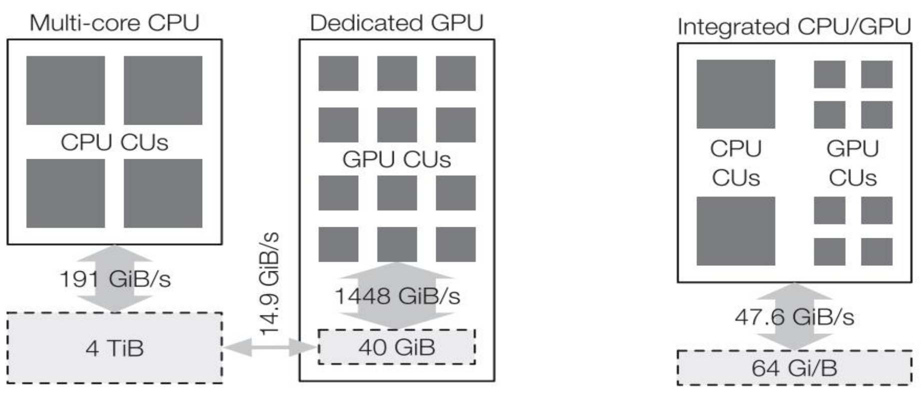


Figure 1.1.1 : Memory size and bandwidth for a multi-core CPU and a dedicated GPU connected over PCIE 3.0(left) and integrated CPU/GPU (right)

1.1.4 GPU Integration

In the traditional setup, GPUs have functioned as dedicated processors connected over a system bus, typically linked to CPUs through a PCI Express (PCIe) 3.0 bus with theoretical bandwidth up to 14.9 GiB/s. Despite facilitating CPU-GPU communication, this connection poses a substantial performance bottleneck, being significantly slower than the main memory bandwidth of the CPU and even more so than GPU memory. Additionally, the lack of coherence between separate CPU and GPU memory spaces necessitates manual synchronization for shared data structures, introducing complexity. Recent GPU architectures, such as AMD GPUs, address this using PCIe atomics for synchronization, while NVIDIA GPUs support software-assisted memory coherence and system-wide atomics with some runtime overhead. IBM Power9 systems leverage NVLink 2.0 to connect CPUs and NVIDIA GPUs, offering hardware-based cache coherence and atomic operations, eliminating associated overheads and providing a significant speed boost over PCIe 3.0, mitigating data transfer bottlenecks.

An alternative approach involves integrating CPUs and GPUs on the same die, exemplified by the AMD Ryzen 7 Pro 4750G. This integration brings several advantages over accessing a dedicated GPU over a system bus. Firstly, both CPU and GPU can directly access main memory, eliminating the need for data copying and expanding the GPU's memory space. Secondly, memory accesses by the CPU and GPU are coherent, enabling simultaneous access and modification of shared data structures through system-wide atomics, fostering fine-grained cooperation and simplified implementations. Thirdly, the close connection in integrated processors results in a superior power/performance ratio compared to dedicated GPUs. However, integrated CPU/GPU processors face drawbacks, including lower memory bandwidth leading to increased memory stalls and reduced performance for memory-bound kernels. Competition for main memory access between the CPU and GPU, coupled with potential saturation of the shared memory subsystem, can further impact performance. In summary, integrated CPU/GPU processors exhibit less raw performance compared to combining a dedicated GPU with a conventional CPU, highlighting the advantages and disadvantages of both integration schemes.

1.1.5 GPGPU Programming Model

Graphic Processing Units (GPUs) employ a distinct programming model, separate from CPUs, enabling the creation of scalable parallel programs. Here GPU hardware is conceptualized as an abstract parallel processor, directing program execution and workload partitioning to achieve scalable parallelism. CUDA and OpenCL are well-known implementations, with CUDA designed specifically for NVIDIA GPUs, prioritizing user-friendliness and performance. In contrast, OpenCL functions as an open standard, platform-independent framework suitable for various GPU architectures, CPUs, and accelerators.

OpenCL is an open standard programming framework that facilitates scalable parallel programming for GPUs, CPUs, and accelerators. Designed by the Khronos Group, it provides a platform-agnostic approach, allowing developers to write parallel programs that can run on various hardware architectures. OpenCL plays a vital role in enabling the effective utilization of GPUs for parallel computing tasks beyond traditional graphics processing.

CUDA, crafted by NVIDIA, stands as a specialized programming framework dedicated to their GPUs. It streamlines the complexity of parallel programming, prioritizing ease of use and performance optimization. Serving as a powerful tool, CUDA empowers developers to tap into the parallel processing potential of NVIDIA GPUs, making it a cornerstone for the effective implementation of parallel programs designed specifically for NVIDIA hardware.

1.2 Replacement Strategies in CPU with GPU

Effective management of the last-level cache (LLC) in chip multi-processors is crucial for optimizing the performance of individual applications and overall system throughput. Contemporary caches, predominantly utilizing recency-friendly Least Recently Used (LRU) approximations, face challenges in scenarios where high cache-demanding applications monopolize cache space, exemplified by streaming applications. Existing solutions have proposed various LLC management mechanisms, such as logical partitioning and pattern filtering, to address LRU-related issues. However, these mechanisms may not be directly applicable to CPU-GPU heterogeneous architectures due to the distinctive characteristics of GPGPU applications. GPGPU applications often leverage massive multi-threading to tolerate memory latency, making caching a secondary remedy. Furthermore, the differing cache access frequencies between CPU and GPGPU applications are not adequately addressed by existing mechanisms, leading to biased cache policies favoring applications with more frequent accesses. To overcome these challenges, we propose two novel policies, MLP-SLCP and MLP-DRRIP, built on Spatial Locality Cache Partitioning (SLCP) and Dynamic Re-Reference Interval Prediction (DRRIP). MLP-SLCP and MLP-DRRIP integrate core sampling and cache block lifetime normalization to adaptively manage cache policies based on the cache behavior and access rates of GPGPU applications. Inspired by Utility-based Cache Partitioning (UCP) and

Re-Reference Interval Prediction (RRIP), our proposed mechanisms aim to enhance cache efficiency in CPU-GPU heterogeneous architectures, addressing the unique challenges posed by GPGPU applications.

1.2.1 Core Sampling

In examining core behavior, each core exhibits consistent progress concerning retired instructions. Core sampling employs distinct policies on individual cores, periodically gathering samples to assess policy efficacy. For instance, to evaluate the impact of cache on performance, Core Sampling directs one core (Core-POL1) to use the LRU insertion policy and another core (Core-POL2) to adopt the MRU insertion policy. After a designated period, the Core Sampling Controller (CSC) collects performance metrics, such as retired instructions, from each core and compares results. Significant performance disparities between Core-POL1 and Core-POL2 suggest a cache behavior impact on application performance. If the performance difference is negligible, caching proves unbeneficial. Based on these results, the CSC makes decisions in the LLC (cache insertion or partitioning), and other cores conform to this determination. Core sampling shares similarities with set dueling, approximating overall cache behavior by sampling a subset of the cache with high probability.

When Core Sampling operates alongside cache partitioning, the influence of different policies on a GPGPU application is confined to its dedicated space within the partition. For example, if a GPGPU application utilizes only one way while CPU applications occupy the remaining ways, sampling policies affect only the GPGPU application's allocated way. In such cases, distinctions between MRU and LRU insertion policies disappear. Consequently, Core Sampling configures Core-POL1 to bypass the LLC through cache partitioning.

1.2.2 Cache Block Lifetime Normalization

Cache Block Lifetime Normalization addresses the inherent distinctions in cache access frequencies between GPGPU and CPU applications. GPGPU applications, characterized by their abundant Thread-Level Parallelism (TLP), exhibit continuous and frequent cache accesses, unlike memory-intensive CPU applications with limited TLP, resulting in less frequent cache accesses. To bridge this gap, we discern access rate differences by monitoring cache accesses from each application. At regular intervals, we compute the access ratio between applications. If the ratio surpasses the predefined threshold (Txs), it is stored in a 10-bit register, XSRATIO. Conversely, when the ratio falls below Txs, XSRATIO defaults to 1. With values exceeding 1, TAP policies utilize XSRATIO to ensure comparable cache residency times for both CPU and GPGPU applications. This adaptive normalization mechanism aims to mitigate the order-of-difference in cache access frequencies, enhancing the equitable treatment of diverse application types within the cache.

1.3 MLP Based Shared LLC Strategies

1.3.1 MLP-SLCP

MLP-SLCP, inspired by SLCP [1], presents an innovative dynamic cache partitioning strategy crafted specifically for the intricacies of heterogeneous CPU-GPU workloads. Unlike its predecessor UCP, designed exclusively for CPU workloads, SLCP recalibrates its cache partition periodically to align with the evolving system behaviour. The mechanism maintains an LRU stack for each sampled set and tracks hit counters for every way across all sampled sets for individual applications. With each cache hit, the corresponding hit counter registers an increment. At the end of a designated period, the partitioning algorithm iterates until all cache ways are allocated based on the marginal utility derived from the remaining ways and hit counters. In contrast to UCP, which exhibits a bias toward GPGPU applications, MLP-SLCP dynamically adapts cache allocations following insights from core sampling. When core sampling indicates minimal caching benefits for a GPGPU application, MLP-SLCP allocates a greater share of cache ways to CPU applications. Furthermore, MLP-SLCP adjusts hit counters for GPGPU applications experiencing substantially higher cache accesses than their CPU counterparts. Two critical modifications are introduced to SLCP's partitioning algorithm within MLP-SLCP: restricting the allocation to one way for GPGPU applications with limited caching benefits and adjusting hit counters by dividing them by the periodically set XSRATIO register value, established through cache block lifetime normalization.

1.3.2 MLP-DRRIP

The Dynamic Re-Reference Interval Prediction (DRRIP) [6] mechanism, which serves as the underpinning for MLP-DRRIP, supplanting RRIP. DRRIP dynamically oscillates between two cache insertion strategies, specifically Static-DRRIP (SDRRIP) and Bimodal-DRRIP (BDRRIP), effectively sifting through thrashing patterns. The essence of DRRIP lies in the Re-Reference Prediction Value (RRPV), denoting the insertion position with an n-bit register per cache block for the LRU counter. An RRPV of 0 designates the Most Recently Used (MRU) position, while an RRPV of n2 -1 signifies the Least Recently Used (LRU) position. SDRRIP consistently inserts incoming blocks with an RRPV of 2n -2, considered the optimal insertion position within the range of 0 to 2n -1. Transitioning to MLP-DRRIP, two pivotal challenges are addressed: 1) addressing scenarios where a GPGPU application necessitates no additional cache space and 2) mitigating interference caused by a GPGPU application with markedly heightened access frequencies. In the presence of either or both issues, MLP-DRRIP mandates the adoption of the BDRRIP policy for GPGPU applications. This decision is grounded in BDRRIP's inclination to enforce a briefer cache lifetime than SDRRIP for each block. Moreover, GPGPU blocks hitting the cache will not be promoted, and when replacement is imminent, GPGPU blocks take precedence over CPU blocks. In pseudo-Least Recently Used (LRU) approximations like DRRIP, where multiple cache blocks can coexist in LRU positions simultaneously, MLP-DRRIP prioritizes GPGPU blocks over CPU blocks for replacement.

2. Literature Review

[1] introduces Core Sampling and TAP Policies as a combined approach to improve cache management in heterogeneous architectures. Core Sampling analyses core behaviour by employing distinct policies on individual cores, providing insights into the impact of cache on performance. The Core Sampling Controller makes decisions in the Last Level Cache (LLC), influencing cache insertion, partitioning, and overall system performance. The paper also addresses cache behaviour in GPGPU applications within dedicated cache spaces and proposes Cache Block Lifetime Normalization to adjust cache residency times based on access rate differences. TAP Policies, including TAP-UCP and TAP-RRIP, dynamically adapt cache partitions and replacement policies based on Core Sampling results, optimizing cache utilization in CPU-GPGPU coexistence. The most renowned and regarded work on is done by Jaleel A. et al [6] who proposed cache replacement using re-reference interval prediction models. The basic LRU replacement algorithm can only predict the near-immediate re-reference interval of a cache block but it works poorly when the actual re-reference interval of a cache block is in the distant future. Even adaptive insertion policies [7] cannot preserve the blocks when the re-reference interval of a cache block contains mixed access patterns. To solve this, they proposed Static Re-reference Interval Prediction (SRRIP) policy which predicts the re-reference interval of a missing cache block to be an intermediate re-reference value which can be anywhere between the near-immediate re-reference interval and the distant re-reference interval. The authors further proposed two SRRIP policies: SRRIP Hit Priority (SRRIP-HP) which predicts the re-

reference interval of cache block based on hits and SRRIP Frequency Priority (SRRIP-FP) which is frequency based. They show that SRRIP-HP generally outperforms SRRIP-FP. The SRRIP models are scan-resistant. As an improvement to SRRIP-HP, the authors also propose Dynamic Re-reference Interval Prediction (DRRIP) policy which, being a scan-resistant policy, is also thrash-resistant as it dynamically determines whether to use SRRIP which is scan-resistant or BRRIP (Bimodal Re-reference Interval Prediction) which is thrash-resistant. Most cited work on adaptive insertion policies mentioned earlier was done by Qureshi et al [7]. They proposed LRU Insertion Policy (LIP) which places the block in LRU position in place of MRU position as LRU replacement policy used to do due to which it protects from thrashing and gives optimal hit-rate for the working set which have a cyclic reference pattern. They also proposed Bimodal Insertion Policy (BIP) which is the optimized version of LIP. This policy adapts to the changing working set while also providing the protection against thrashing along with fewer miss rates. They also proposed Dynamic Insertion Policy (DIP) which dynamically chooses between LIP and BIP depending upon which of them give the higher hit rates. Implementation of DIP is done by implementing both LIP and BIP in separate tag directories and keeping track of which of them provides fewer miss rates and the main tag directory automatically chooses the best among them. [2] addresses the issue of unfair cache space utilization between CPU and GPU cores in heterogeneous multicore processors. It proposes a new method called cache partitioning combined with adaptive replacement policy to mitigate this problem. The method aims to improve CPU application performance by up to 33% and overall system performance by up to 19% with minimal impact on GPU application performance. The method also proposes to use LRU for CPU access type and tree-based PLRU for GPU access types in its adaptive replacement method. The Tree based LRU has been proposed and described in [3]&[4]. [5] investigates fairness in cache sharing among threads in a chip multiprocessor (CMP) architecture, an area that has received limited attention in prior research. It introduces and evaluates five cache fairness metrics and proposes static and dynamic L2 cache partitioning algorithms to optimize fairness. The study reveals a strong correlation between certain fairness metrics and execution-time fairness and demonstrates that optimizing fairness can lead to increased throughput. The proposed algorithms significantly improve fairness while increasing throughput by 15% compared to non-partitioned shared caches. Additionally, the paper discusses hardware support for partitionable caches and distinguishes its partitioning algorithms by focusing on fairness optimization and compatibility with pseudo-LRU replacement algorithms commonly used in L2 caches. [8] investigates the use of adaptive insertion policies to manage shared caches in chip multiprocessors (CMPs). It introduces the Thread-Aware Dynamic Insertion Policy (TADIP) to address the limitations of the Dynamic Insertion Policy (DIP) in shared cache environments. The evaluation with multi-programmed workloads for different core CMPs demonstrates that TADIP significantly improves overall throughput compared to the baseline LRU policy and outperforms DIP. The paper also categorizes applications based on their cache behavior and proposes scalable mechanisms, TADIP-Isolated and TADIP-Feedback, to further enhance cache performance. The study concludes by highlighting the applicability of adaptive insertion to optimize various metrics and the ongoing exploration of extensions for different access streams. [9] introduces Utility-Based Cache Partitioning (UCP), a low-overhead, runtime mechanism that dynamically partitions a shared cache between multiple applications based on the reduction in cache misses each application is likely to obtain for a given amount of cache resources. It proposes a novel, cost-effective hardware circuit to monitor each application at runtime and uses the collected information to allocate cache resources. The evaluation with 20 multi-programmed workloads demonstrates that UCP improves the performance of a dual-core system by up to 23% and on average by 11%

over LRU-based cache partitioning. To implement UCP's decisions, the baseline LRU policy is augmented to enable way partitioning. This involves adding a bit to the tag-store entry of each block to identify the core that installed the block in the cache. The paper also proposes the Lookahead Algorithm as a scalable alternative for partitioning decisions when a large number of applications share a highly associative cache. The paper's contributions include the introduction of a low hardware overhead utility monitoring circuit and the Lookahead Algorithm, both of which enhance cache partitioning decisions. The authors also discuss potential extensions of UCP to SMT processors and the incorporation of utility information for prefetching and CPI estimation. Additionally, they highlight the applicability of UCP to multithreaded workloads and the potential for implementing execution-time fairness without requiring profile information. The paper concludes by outlining future work on exploring these extensions. [9] focuses on enhancing the efficiency of the last-level cache (LLC) in CPU-GPU heterogeneous multi-core architectures by dynamically adjusting the cache replacement policy based on the priority of CPU and GPU applications. The study acknowledges the imbalance in cache demand between CPU and GPU applications and proposes a novel strategy to address this issue. Specifically, the paper introduces a dynamic policy-switching mechanism between the Least Recently Used (LRU) and Least Frequently Used (LFU) policies, considering both the time and frequency of cache block access in the LLC. The proposed method aims to optimize system performance by giving higher priority to CPU applications in the cache queue, reflecting the higher sensitivity and demand for cache blocks compared to GPU applications. By modifying the cache block priorities based on the type of application accessing the cache, the research demonstrates that the optimized cache replacement policy significantly impacts the performance of the heterogeneous multi-core architecture. The experimental results highlight the effectiveness of the proposed cache replacement optimization method, emphasizing its significance for further research in

CPU-GPU heterogeneous multi-core architectures. The paper's contribution lies in providing a foundation for the dynamic adjustment of cache replacement policies in such architectures, offering valuable insights for future studies on system performance and power consumption in CPU-GPU heterogeneous multi-core environments. [11] explores the use of Dynamic Cache Partitioning (DCP) to improve traditional eviction policies in multithreaded architectures. It addresses challenges related to thread-level parallelism (TLP) and resource sharing in paradigms like simultaneous multithreading (SMT) and chip multiprocessing (CMP). Unlike existing DCP algorithms, the proposed method considers the impact of cache misses on performance, distinguishing between clustered and isolated misses based on Memory Level Parallelism (MLP) distribution. The algorithm significantly improves throughput over traditional policies and outperforms previous DCP proposals in a four-core architecture. The practical implementation incurs minimal hardware cost, making it a feasible enhancement for multithreaded systems. Current cache designs face inefficiencies in accommodating the working set requirements of modern applications due to fixed geometries. Many applications exhibit short block lifetimes and only moderate spatial locality, resulting in a substantial portion of unused words within cache blocks. This leads to significant downsides, such as an increased miss rate, wastage of on-chip bandwidth, and energy consumption in the cache hierarchy. Addressing these challenges, [12] proposes Amoeba-Cache which introduces a ground-breaking design that supports a variable number of cache blocks with different granularities. Notably, Amoeba-Cache eliminates the conventional tag array, treating the storage array as adaptable between tags and data. This innovative approach allows the cache to reclaim space from unused words within blocks for additional tag storage, enabling support for a variable number of tags and, consequently, blocks. The adaptability of Amoeba-Cache extends to dynamically adjusting cache line granularities based on the spatial locality of applications, optimizing for different data objects and access phases. Comparative analysis indicates that, when compared to fixed granularity caches, Amoeba-Cache consistently reduces miss rates at both L1 and L2 levels, L1–L2 miss bandwidth, on-chip memory hierarchy energy consumption, and significantly improves overall performance. This adaptability positions Amoeba-Cache as a versatile solution capable of addressing the diverse characteristics of various applications. [15] presents Elastic Cache, an innovative solution to enhance L1 cache efficiency in GPUs by addressing challenges associated with irregular memory access patterns. Traditional wide L1 cache lines in GPUs may not be optimally utilized in scenarios involving frequent branch and memory divergences. Elastic-Cache efficiently manages both fine- and coarse-grained L1 cache-line scenarios by allowing the storage of 32- or 64-byte words in non-contiguous memory spaces within a single 128-byte cache line. Crucially, it achieves this without compromising the cache size, as it stores chunk-tags in unused shared memory instead of using dedicated arrays or data arrays. This approach not only makes Elastic-Cache suitable for GPUs with limited per-thread cache size but also distinguishes it from other methods like Sector-Cache. Elastic-Cache's ability to adapt to irregular memory access patterns significantly improves efficiency, as demonstrated by a 58% average improvement in IPC over baseline GPUs in applications exhibiting such access patterns. It achieves this by leveraging unused shared memory to store tags for fine-grained cache-line management, complemented by a conventional tag array for coarse-grained cache accesses. The novel features of Elastic-Cache make it a promising solution for optimizing L1 cache performance in GPU architectures. [16] examines dynamic cache partitioning (DCP) algorithms and runtime mechanisms aimed at improving the performance of shared last-level caches (L2) in multi-core processors. The DCP algorithm assigns costs to L2 accesses based on their impact on performance, favoring clustered misses over isolated misses. By minimizing the total cost for all running threads, the algorithm identifies the optimal L2 cache partition for multi-core workloads, achieving performance similar to a 50% larger cache. Additionally, a runtime mechanism dynamically divides shared L2 caches in chip multiprocessor (CMP) scenarios, considering the memory-level parallelism (MLP) of each L2 access. This approach delivers significant performance enhancements over traditional policies, with gains of up to 63.9% over least recently used (LRU) and up to 15.4% over previous proposals in a four-core architecture. Furthermore, a sampling technique is introduced to reduce hardware cost in terms of storage to less than 1% of the total L2 cache size, with minimal impact on throughput. [17] delves into the complexities of running multiple applications concurrently on multi-core and heterogeneous architectures, particularly focusing on managing shared resources like the on-chip Last Level Cache (LLC). It underscores the importance of LLC management in mitigating delays before accessing the slower main memory, especially in multi-core processors. The paper highlights existing LLC management strategies such as UCP and RRIP for multi-core systems and their adaptations for heterogeneous architectures. It also emphasizes the need to extend replacement policies in GPGPU-Sim to evaluate new techniques in heterogeneous systems. Additionally, it underscores the importance of LLC management in GPGPU applications due to their high memory access rates and potential impact on concurrent CPU applications. Furthermore, the paper addresses the challenges posed by shared resources in heterogeneous architectures, including shared LLC, DRAM controller, front side bus, and prefetching hardware. It notes the shift towards private prefetchers and more advanced on-die DRAM controllers in recent processor designs, emphasizing the significance of LLC management in enhancing application performance and overall system throughput. In [18], the research community has extensively explored the capabilities of graphic processing units (GPUs) in database systems due to their substantial computational power. The integration of CPUs and GPUs in a heterogeneous query processing system presents challenges such as workload distribution, addressing data transfer bottlenecks, and ensuring efficient support for multiple processors. The survey introduces a classification scheme for techniques addressing these challenges, providing insights and pinpointing open research problems. While query processing systems effectively utilize GPUs to enhance performance, there is a distinct emphasis on dedicated GPUs, often overlooking integrated GPUs. Given the differing optimization requirements for dedicated and integrated GPUs, findings for dedicated GPUs may not universally apply. Relational heterogeneous query processors typically lean towards CPU-centric implementations, necessitating further research to determine the most efficient processing model on GPUs and explore combinations of different models on CPUs and GPUs. The ongoing innovation in GPU hardware capabilities offers exciting research opportunities, particularly in the integration of machine learning with relational query processing. This dynamic landscape calls for sustained exploration within the research community.

MacSim[13] is a trace-driven, cycle-level heterogeneous architecture simulator that meticulously models architectural behaviours, encompassing detailed pipeline stages, multi-threading, and memory systems. The simulator accommodates both heterogeneous and homogeneous Instruction Set Architectures (ISA), supporting x86 and PTX trace instructions, which are internally converted into RISC-style micro-ops (uop) for simulation. To extend its compatibility to other ISAs such as ARM, a frontend simulator or functional emulator is employed. The simulator utilizes knob variables defined in def/*.param.def files to control simulation and architectural parameters. These knob definitions are automatically converted during the build process into C++ source code, which is then included in the compilation of the MacSim binary. By adjusting parameter values for the knob variables, MacSim can be configured to simulate various CPU, GPU, and heterogeneous architectures. MacSim's architecture includes a five-stage pipelined processor with adjustable pipeline depth, encompassing Fetch, Decode, Schedule and Execute (including Memory), and Retire stages. The simulator employs a common Process Manager/Thread Scheduler for both CPU threads and GPU warps. For each simulated application, the Process Manager creates a process and associated threads or warps, assigning dedicated cores based on the simulation configuration.

[14] presents Pin, an advanced software instrumentation system tailored for program analysis tasks such as profiling, performance evaluation, and bug detection. Pin addresses the critical need for robust and versatile instrumentation tools by offering a solution that is user-friendly, portable, transparent, and efficient. The system introduces Instrumentation tools, referred to as Pintools, which are written in C/C++ using Pin's comprehensive API. Embracing the ATOM model, Pin allows tool developers to analyse applications at the instruction level without requiring an intricate understanding of the underlying instruction set. Pin employs dynamic compilation to instrument running executables, demonstrating significantly enhanced performance compared to similar tools. The architecture-independent design of Pin's API facilitates source compatibility across diverse architectures, supporting ISAs such as IA32, EM64T, Itanium, and ARM. The paper provides a detailed overview of Pin's architecture, outlining key components like the virtual machine (VM), code cache, and the instrumentation API invoked by Pintools. Specific aspects, including Pin's JIT compiler, trace linking mechanism, register re-allocation, and optimizations for instrumentation performance, are thoroughly discussed. [14] also delves into the organization of Pin's source code, emphasizing strategies employed to minimize development effort through efficient code sharing among various architectures.

3. Research Objective

We have identified following objectives for our thesis based on previous works to maximize the utilization of the shared LLC with different standard benchmarks workloads and accelerate overall performance of a multi-core processor in Heterogeneous ISA.

Objective-1: To enhance the performance of Shared Last Level Cache based on Memory Level Parallelism based Replacement Strategies for Heterogenous CPU-GPU Multi-Core Processor Architecture

4. Simulation Environment

According to [1] the classification of GPGPU application types is based on distinct characteristics. Type A applications are considered ideal, showcasing low CPI (Cycles Per Instruction), low MPKI (Misses Per Kilo Instructions), and medium overall performance. Type B applications exhibit high CPI, extremely high MPKI, and an average performance level. Type C applications, in addition to being less than ideal in terms of CPI and having low MPKI, are highlighted for their cache-friendly nature, contributing to extremely high performance. Type D applications are close to ideal in terms of CPI, with reduced MPKI and higher-than-average performance. Lastly, Type E applications have higher CPI than Type B, high MPKI, and an average performance level. This classification provides a nuanced understanding of the GPGPU application landscape, considering factors such as cache-friendliness and performance metrics. This is shown in Table 1.3.1.

GPGPU Types	**CPI**	**MPKI (Misses Per Kilo Instructions)**	**Performance**
A	Ideal	Low	Medium
B	High	Extremely High	Average
C	Less than Ideal	Low	Extremely High
D	Close to Ideal CPI	Reduced	Higher than average
E	Higher than that of type B	High	Average

Table 1.3.1: Classification of GPGPU Application Types based on their characteristics

Building upon this, the first table outlines the specifications of a heterogeneous CPU-GPU simulation environment designed to accommodate these GPGPU application types. The CPU component offers flexibility with a choice between 4 or 8 cores and various L3 cache options.

Meanwhile, the GPU features a 6-core design with a 2-wide 8-SIMD architecture. The simulation configuration are given in Table 1.3.2. The CPU benchmarks encompass cache-aware, streaming, and CPU-intensive applications, including mcf, bzip2, omnetpp, gcc, bwaves, libquantum, gobmk, and perlbench. On the GPU side, benchmarks are categorized into GPGPU-A to GPGPU-E, covering a diverse range of applications such as fastwalsh, dxtc, volumerender, bicubic, convtex, quasirandom, blackscholes, mergesort, sobolqrng, sobelfilter, raytracing, reduction, histogram, montecarlo, and bfs. The combination of CPU and GPU benchmarks, along with the specified architecture, allows for a comprehensive simulation environment provided with 32 workloads. The incorporation of cache-friendly Type C applications further enhances the overall performance of the simulation, making it well-suited for a wide range of heterogeneous computing scenarios. This is shown in Table 1.3.3.

CPU	4 cores, 8 cores
GPU	6 core, 2-wide 8-SIMD
L3	8-way 16MB, 16-way 32MB

Table 1.3.2: Simulation Configuration

Architecture	**Benchmark**	**Application Program Mix**
CPU	Cache-Aware	mcf, bzip2, omnetpp, gcc
	Streaming	bwaves, libquantum
	CPU Intensive	gobmk, perlbench
GPU	GPGPU-A	fastwalsh, dxtc, volumerender
	GPGPU-B	bicubic, convtex

	GPGPU-C	quasirandom, blackscholes, mergesort, sobolqrng, sobelfilter, raytracing
	GPGPU-D	reduction, histogram
	GPGPU-E	montecarlo, bfs

Table 1.3.3: CPU and GPU Benchmarks

Different types of workloads divided have a tendency to perform better by having a preference. For workload type A, the recommendation is to perform computations on the CPU when using SLCP, while the GPGPU is favoured when employing DRRIP. Workload type B is recommended for GPGPU processing under both replacement policies. Similarly, workload type C is advised for GPGPU computation in both SLCP and DRRIP scenarios. In contrast, workload type D is recommended for CPU computation with SLCP and GPGPU processing with DRRIP. Finally, workload type E is suggested for CPU processing under both SLCP and DRRIP. These recommendations provide insights into the preferred computing unit (CPU or GPGPU) for specific workloads based on the characteristics of GPGPU application types and the chosen replacement policies, offering guidance for optimizing task allocation in the heterogeneous

Workloads	**Recommended Application type**		
CPU+	GPGPU Type	SLCP	DRRIP
	A	CPU	GPGPU
	B	GPGPU	GPGPU
	C	GPGPU	GPGPU
	D	GPGPU	CPU
	E	CPU	CPU

CPU-GPU simulation environment. This is shown in Table 1.3.4.

Table 1.3.4: Recommended Processor Type for different application

The above configuration were simulated in MacSim simulator by generating x86 traces using Pin Tool. MacSim internally converts the x86 trace instructions into RISC-style micro-operations (uops) for simulation as shown in Figure 1.3.1.

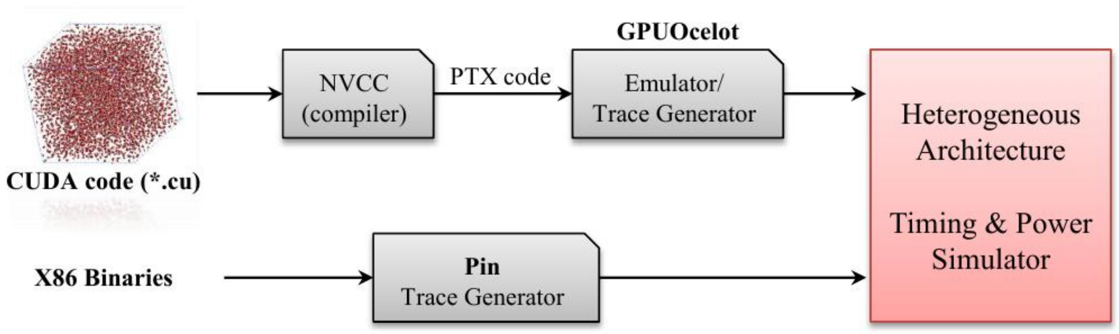


Figure 1.3.1 : Heterogeneous CPU-GPU Simulation with MacSim Simulator and Trace Generation with Pin Tool

The target architecture for 4-core CPU with 6-core GPU is Figure 1.3.2.

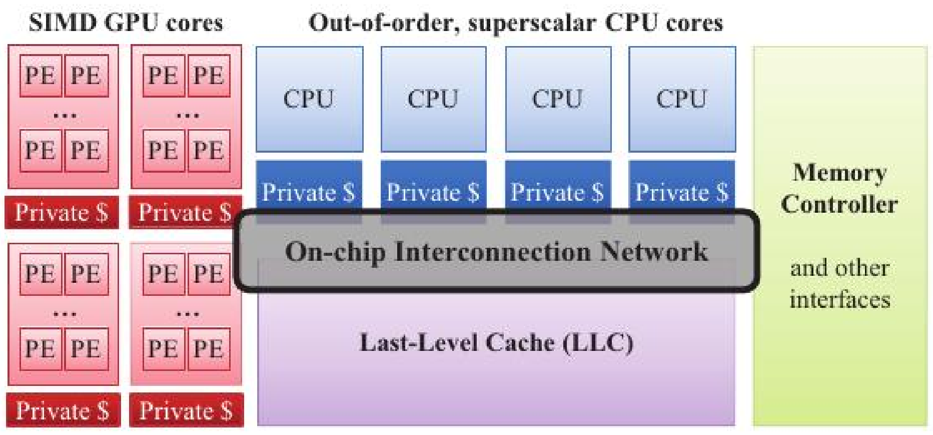


Figure 1.3.2 : Target 4-core CPU, 6-core GPU Architecture

5. Implementation

5.1 Introduction

In the pursuit of optimizing cache replacement policies for heterogeneous CPU-GPU simulation environments, this paper introduces two innovative strategies, MLP-SLCP and MLP-DRRIP, built upon the established frameworks of Spatial Locality Cache Partitioning (SLCP) and Dynamic Re-Reference Interval Prediction (DRRIP). The study addresses the inherent challenges arising from the unique characteristics of CPU and GPGPU applications when sharing the Last-Level Cache (LLC). Unlike traditional memory-intensive CPU benchmarks that leverage caching to mitigate latency, numerous GPGPU applications rely on a combination of thread-level parallelism (TLP) and caching to obscure memory latency. This nuanced distinction poses a significant challenge in devising cache replacement policies that cater to the divergent behaviors of CPU and GPGPU workloads. Through extensive simulations spanning various core configurations and cache sizes, our proposed MLP-enhanced policies demonstrate substantial speedups, with MLP-SLCP showcasing an average improvement of 38.8% over SLCP and MLP-DRRIP exhibiting an average enhancement of 31.3% over DRRIP.

5.2 Speedup Analysis

5.2.1 Performance Metrics

We have calculated speedup of the configurations based on CPI of the system all over 32 workloads. The final speedup is the geometric mean over all 32 workloads. We have used LRU for our baseline comparison in speedup.

$$Spspeedup_i = \frac{CPI_{i_{LRU}}}{CPI_i} \tag{1}$$

$$Speedup = \sqrt[n]{\sum_{i=0}^{n-1} speedup_i}$$

(II)

5.2.2 Results

The results obtained by varying number or cores are:

4-core					
Workloads	**CPI_LRU**	**CPI_SLCP**	**CPI_MLP_SLCP**	**CPI_DRRIP**	**CPI_MLP_DRRIP**
w0	0.948492421	0.890456239	0.746956883	0.802794054	0.636693881
w1	0.916440883	0.897077197	0.738764313	0.816734657	0.604078592
w2	0.915898897	0.914390271	0.71190358	0.837426134	0.637097453
w3	0.951354964	0.850277189	0.792456196	0.805173496	0.665423551
w4	0.874641425	0.766168204	0.711062721	0.836981895	0.631342677
w5	0.939472478	0.781208064	0.792235116	0.803925796	0.601856625
w6	0.912564819	0.785738844	0.785081498	0.825767766	0.674393362
w7	0.907414344	0.92438802	0.745785602	0.830288251	0.644803523
w8	0.982901716	0.835196122	0.747820668	0.832434395	0.670892097
w9	0.949962954	0.768398513	0.76716953	0.827787721	0.667403431
w10	0.949939105	0.764030185	0.752191973	0.800453009	0.661121149
w11	0.984876588	0.837421953	0.750958115	0.845059242	0.621014975
w12	0.820150733	0.9102971	0.79155047	0.817166377	0.667111363
w13	0.952480713	0.751065635	0.786484164	0.802082855	0.685462626
w14	0.967020741	0.947151483	0.701076527	0.816298723	0.650749197
w15	0.973158703	0.941704589	0.797373565	0.808002785	0.64236407
w16	0.896691915	0.759174106	0.79266278	0.833879541	0.646965769
w17	0.816697023	0.907894947	0.797333702	0.827525086	0.64116388
w18	0.800502428	0.89926082	0.774952416	0.837469679	0.663499908
w19	0.824882991	0.851433763	0.701531872	0.843789133	0.697725718
w20	0.862018572	0.818659631	0.739866056	0.801699134	0.612904309
w21	0.978982806	0.825002918	0.797952428	0.84914667	0.672646717
w22	0.913412913	0.882402115	0.737175547	0.8281931	0.612164173
w23	0.847207048	0.827169829	0.735477612	0.839300124	0.691765094
w24	0.886838462	0.820098291	0.766929602	0.84688136	0.662667619
w25	0.978028171	0.817121368	0.78756074	0.834550202	0.668533858
w26	0.931112471	0.906871099	0.765768769	0.815056107	0.607185784
w27	0.924124374	0.810290941	0.763058427	0.820221894	0.665976987
w28	0.987329013	0.903266663	0.718574436	0.831722356	0.647836026
w29	0.953087886	0.828695603	0.709095852	0.818790958	0.687264639
w30	0.840218247	0.78782548	0.775099678	0.800855244	0.640623355
w31	0.958565945	0.796737759	0.749288094	0.834821219	0.657137499
Mean	0.914363597	0.84045194	0.756924069	0.824691854	0.651131404
Speedup	1	1.087942751	1.207999104	1.108733634	1.404268926

Table 5.2.1 : CPI of various mechanisms over 4-core architecture

8-core					
Workloads	**CPI_LRU**	**CPI_SLCP**	**CPI_MLP_SLCP**	**CPI_DRRIP**	**CPI_MLP_DRRIP**
w0	0.79586168	0.742529067	0.539522138	0.796839885	0.459824396
w1	0.841155445	0.772426042	0.533805242	0.741608893	0.436930139
w2	0.929962336	0.858312261	0.544427806	0.678941878	0.473094168
w3	0.925987662	0.79143493	0.546142567	0.795600274	0.485474751
w4	0.781552205	0.754717169	0.52700569	0.735727221	0.422972566
w5	0.872608327	0.758126893	0.52470691	0.714463336	0.405136752
w6	0.820237636	0.896180834	0.54410886	0.7586369	0.486423199
w7	0.819962168	0.878698556	0.524669632	0.803389909	0.486136039
w8	0.843838604	0.882525761	0.543301611	0.77408156	0.442139977
w9	0.794062926	0.836620783	0.572954785	0.834655831	0.431791312
w10	0.945161618	0.888368561	0.529340254	0.661271544	0.415161115
w11	0.905276784	0.847663603	0.518934713	0.845533322	0.437096556
w12	0.901848286	0.745688904	0.5907404	0.713690905	0.428662396
w13	0.836456028	0.749413779	0.563073857	0.831366943	0.48035358
w14	0.920724294	0.792820605	0.562158649	0.652700971	0.489637899
w15	0.901917836	0.76607336	0.555716756	0.691853468	0.411896705
w16	0.789059998	0.856194842	0.59831335	0.754808547	0.462692581
w17	0.826295195	0.875867608	0.522487007	0.654026666	0.446353743
w18	0.778258041	0.844126537	0.540360217	0.678796403	0.454608549
w19	0.833335193	0.758253751	0.517930548	0.786878837	0.450067813
w20	0.781083699	0.748416508	0.509852339	0.826266245	0.465652744
w21	0.908513186	0.781670541	0.541363558	0.781532941	0.447279822
w22	0.88507087	0.778614803	0.573500207	0.695547644	0.462948308
w23	0.76286274	0.873195122	0.53591079	0.835176547	0.48227532
w24	0.902031797	0.847021636	0.558470503	0.750262549	0.477026838
w25	0.833882931	0.705570657	0.537167289	0.706754321	0.418867247
w26	0.909237036	0.78371868	0.578958966	0.787399386	0.433806265
w27	0.788913044	0.843736722	0.507291712	0.732414212	0.446861492
w28	0.865621458	0.747836614	0.511306724	0.709133508	0.479502763
w29	0.842779989	0.7897765	0.530909921	0.770924018	0.436156524
w30	0.865918023	0.762552625	0.557181457	0.675869071	0.410677611
w31	0.790482069	0.735867021	0.596436536	0.828988822	0.475909342
Mean	0.850115672	0.803078206	0.544559826	0.746339659	0.45035092
Speedup	1	1.058571463	1.561106109	1.139046628	1.887673887

Table 5.2.2 : CPI of various mechanisms over 8-core architecture

The results obtained by varying cache sizes are as follows:

8MB-16-way					
Workloads	**CPI_LRU**	**CPI_SLCP**	**CPI_MLP_SLCP**	**CPI_DRRIP**	**CPI_MLP_DRRIP**
w0	0.948492421	0.890456239	0.746956883	0.867017578	0.700363269
w1	0.916440883	0.897077197	0.738764313	0.88207343	0.664486451
w2	0.915898897	0.914390271	0.71190358	0.904420225	0.700807199

w3	0.951354964	0.850277189	0.792456196	0.869587375	0.731965907
w4	0.874641425	0.766168204	0.711062721	0.903940447	0.694476945
w5	0.939472478	0.781208064	0.792235116	0.86823986	0.662042288
w6	0.912564819	0.785738844	0.785081498	0.891829187	0.741832698
w7	0.907414344	0.92438802	0.745785602	0.896711311	0.709283876
w8	0.982901716	0.835196122	0.747820668	0.899029146	0.737981306
w9	0.949962954	0.768398513	0.76716953	0.894010739	0.734143774
w10	0.949939105	0.764030185	0.752191973	0.864489249	0.727233264
w11	0.984876588	0.837421953	0.750958115	0.912663982	0.683116472
w12	0.820150733	0.9102971	0.79155047	0.882539687	0.733822499
w13	0.952480713	0.751065635	0.786484164	0.866249483	0.754008889
w14	0.967020741	0.947151483	0.701076527	0.88160262	0.715824117
w15	0.973158703	0.941704589	0.797373565	0.872643008	0.706600477
w16	0.896691915	0.759174106	0.79266278	0.900589904	0.711662346
w17	0.816697023	0.907894947	0.797333702	0.893727093	0.705280268
w18	0.800502428	0.89926082	0.774952416	0.904467254	0.729849898
w19	0.824882991	0.851433763	0.701531872	0.911292263	0.76749829
w20	0.862018572	0.818659631	0.739866056	0.865835064	0.67419474
w21	0.978982806	0.825002918	0.797952428	0.917078403	0.739911389
w22	0.913412913	0.882402115	0.737175547	0.894448548	0.67338059
w23	0.847207048	0.827169829	0.735477612	0.906444134	0.760941603
w24	0.886838462	0.820098291	0.766929602	0.914631869	0.728934381
w25	0.978028171	0.817121368	0.78756074	0.901314218	0.735387244
w26	0.931112471	0.906871099	0.765768769	0.880260595	0.667904363
w27	0.924124374	0.810290941	0.763058427	0.885839646	0.732574686
w28	0.987329013	0.903266663	0.718574436	0.898260144	0.712619629
w29	0.953087886	0.828695603	0.709095852	0.884294235	0.755991103
w30	0.840218247	0.78782548	0.775099678	0.864923663	0.704685691
w31	0.958565945	0.796737759	0.749288094	0.901606916	0.722851248
Mean	0.914363597	0.84045194	0.756924069	0.890667202	0.716244544
speedup	1	1.087942751	1.207999104	1.026605217	1.276608114

Table 5.2.3 : CPI of various mechanisms over 8MB-16-way cache

16MB-32way					
Workloads	CPI_LRU	CPI_SLCP	CPI_MLP_SLCP	CPI_DRRIP	CPI_MLP_DRRIP
w0	0.79586168	0.742529067	0.66630984	0.876523873	0.666745374
w1	0.841155445	0.772426042	0.659249474	0.815769782	0.633548701
w2	0.929962336	0.858312261	0.67236834	0.746836066	0.685986544
w3	0.925987662	0.79143493	0.67448607	0.875160302	0.703938389
w4	0.781552205	0.754717169	0.650852027	0.809299943	0.61331022
w5	0.872608327	0.758126893	0.648013033	0.785909669	0.587448291
w6	0.820237636	0.896180834	0.671974442	0.83450059	0.705313638
w7	0.819962168	0.878698556	0.647966995	0.8837289	0.704897256
w8	0.843838604	0.882525761	0.67097749	0.851489716	0.641102967
w9	0.794062926	0.836620783	0.70759916	0.918121414	0.626097402

w10	0.945161618	0.888368561	0.653735214	0.727398698	0.601983617
w11	0.905276784	0.847663603	0.640884371	0.930086654	0.633790006
w12	0.901848286	0.745688904	0.729564394	0.785059995	0.621560475
w13	0.836456028	0.749413779	0.695396214	0.914503638	0.69651269
w14	0.920724294	0.792820605	0.694265932	0.717971068	0.709974953
w15	0.901917836	0.76607336	0.686310193	0.761038815	0.597250222
w16	0.789059998	0.856194842	0.738916987	0.830289402	0.670904242
w17	0.826295195	0.875867608	0.645271454	0.719429333	0.647212928
w18	0.778258041	0.844126537	0.667344868	0.746676043	0.659182396
w19	0.833335193	0.758253751	0.639644227	0.86556672	0.652598329
w20	0.781083699	0.748416508	0.629667638	0.90889287	0.675196479
w21	0.908513186	0.781670541	0.668583994	0.859686235	0.648555742
w22	0.88507087	0.778614803	0.708272755	0.765102409	0.671275046
w23	0.76286274	0.873195122	0.661849825	0.918694202	0.699299214
w24	0.902031797	0.847021636	0.689711071	0.825288804	0.691688915
w25	0.833882931	0.705570657	0.663401601	0.777429753	0.607357508
w26	0.909237036	0.78371868	0.715014323	0.866139325	0.629019085
w27	0.788913044	0.843736722	0.626505264	0.805655634	0.647949163
w28	0.865621458	0.747836614	0.631463804	0.780046858	0.695279006
w29	0.842779989	0.7897765	0.655673752	0.848016419	0.63242696
w30	0.865918023	0.762552625	0.688119099	0.743455978	0.595482535
w31	0.790482069	0.735867021	0.736599122	0.911887705	0.690068546
Mean	0.850115672	0.803078206	0.672531385	0.820973625	0.653008834
Speedup	1	1.058571463	1.26405353	1.035496934	1.30184406

Table 5.2.4 : CPI of various mechanisms over 16MB-32-way cache

5.2.3 Analysis

From the analysis of results from Table 5.2.1 & Table 5.2.2 of replacement policies in a heterogeneous simulation environment with 4-core and 8-core configurations, the speedup comparisons against the LRU policy reveal notable performance improvements. The Spatial Locality Cache Partitioning (SLCP) strategy exhibits a speedup of approximately 8.8% and 5.9% for the 4-core and 8-core configurations, respectively, indicating an enhancement in performance over the LRU policy. The introduction of Memory Level Parallelism (MLP) through MLP_SLCP results in a substantial speedup of 50.2% (8-core), showcasing a significant performance gain compared to SLCP. The Dynamic Re-Reference Interval Prediction (DRRIP) replacement policy provides a speedup of about 10.9% (4-core) and 13.9% (8-core), while the MLP_DRRIP strategy yields remarkable improvements of approximately 40.4% (4-core) and 88.8% (8-core). These results underscore the significance of Memory Level Parallelism in enhancing the efficiency of cache replacement policies in heterogeneous computing environments. This is shown in Figure 5.2.1.

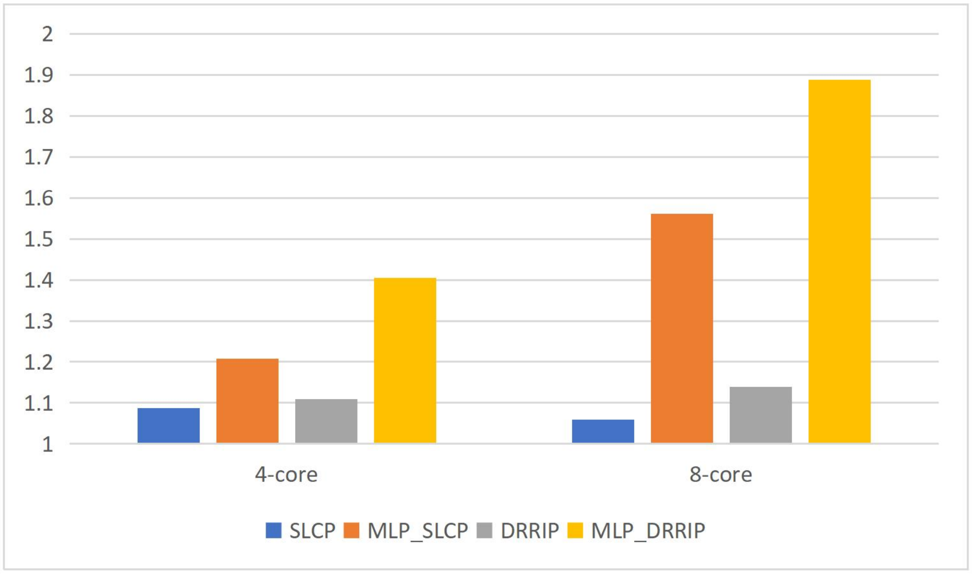


Figure 5.2.1: Speedup Over LRU of different mechanisms

In the exploration of cache replacement policies within a heterogeneous simulation environment, our evaluation extends to varying cache sizes—specifically, 8 MB with a 16-way set-associative configuration and 16 MB with a 32-way set-associative configuration as shown in Table 5.2.3 & Table 5.2.4. The Spatial Locality Cache Partitioning (SLCP) strategy demonstrates a speedup of approximately 8.8% (8MB-16-way) and 5.9% (16MB-32way) compared to the LRU policy, showcasing its effectiveness in both cache size scenarios. The introduction of Memory Level Parallelism (MLP) through MLP_SLCP results in notable speedups of 20.8% (8MB-16-way) and 19.7% (16MB-32way), indicating a consistent performance gain over SLCP. The Dynamic Re-Reference Interval Prediction (DRRIP) replacement policy provides speedups of approximately 2.7% (8MB-16-way) and 3.5% (16MB-32way), while the MLP_DRRIP strategy yields substantial improvements of 27.6% (8MB-16-way) and 27.7% (16MB-32way). These results underscore the nuanced impact of cache replacement policies in heterogeneous computing environments, emphasizing the importance of considering both cache size and replacement strategy for optimal performance which is highlighted in Figure 5.2.2.

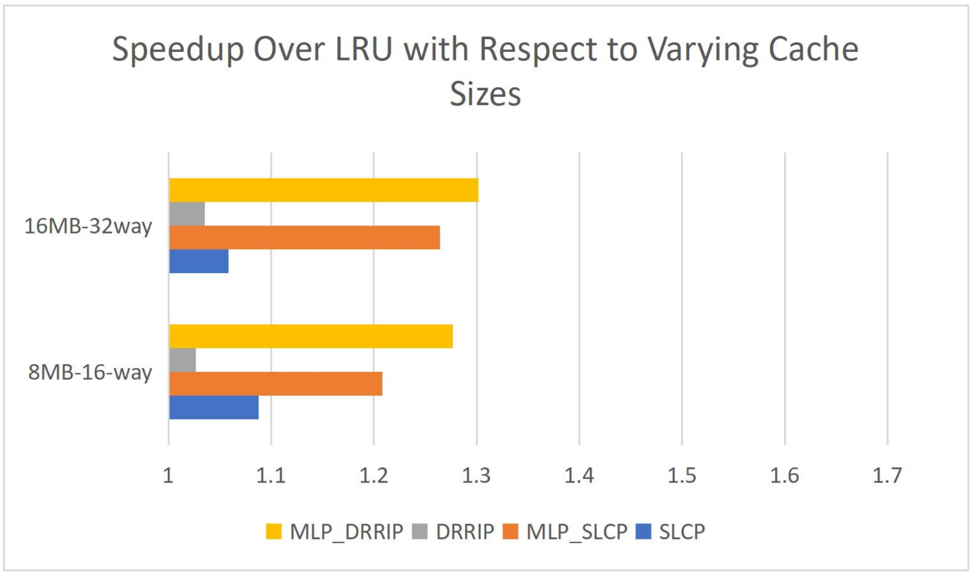


Figure 5.2.2: Speedup Over LRU with Respect to Varying Cache Sizes

6. Conclusion and Future Work

In conclusion, our research has introduced novel cache management strategies, MLP-SLCP and MLP-DRRIP, designed specifically for heterogeneous CPU-GPU simulation environments. These policies address the challenges posed by the distinct characteristics of CPU and GPGPU applications, providing adaptive solutions to optimize cache utilization. MLP-SLCP dynamically adjusts cache allocations based on core sampling insights, ensuring equitable treatment for both CPU and GPGPU applications. On the other hand, MLP-DRRIP tackles interference and cache space utilization challenges by dynamically switching between cache insertion policies for GPGPU applications.

The experimental results showcase significant performance improvements over traditional mechanisms. MLP-SLCP demonstrates an average improvement of 38.8% over SLCP, while MLP-DRRIP exhibits an average enhancement of 31.3% over DRRIP. These percentage improvements underscore the efficacy of our proposed MLP-enhanced policies, emphasizing their potential to enhance cache management in heterogeneous architectures.

As a direction for future work, we aim to extend our research to larger server architectures. Scaling up to more complex systems introduces additional challenges and opportunities for optimizing cache policies. We plan to explore the adaptability and scalability of MLP-SLCP and MLP-DRRIP in the context of larger-scale server environments, considering diverse workloads and architectural configurations. This extension will contribute to a deeper understanding of the applicability and performance gains of MLP-enhanced cache management strategies in broader computing ecosystems.

7. Bibliography

[1] Lee, J., & Kim, H. (2012, February). TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture. In IEEE International Symposium on High-Performance Comp Architecture (pp. 1-12). IEEE.

[2] Fang, J., Liu, S., & Zhang, X. (2017, October). Research on cache partitioning and adaptive replacement policy for cpu-gpu heterogeneous processors. In 2017 16th International Symposium on Distributed Computing and Applications to Business, Engineering and Science (DCABES) (pp. 19-22). IEEE.

[3] Ghasemzadeh, H., Mazrouee, S., & Kakoee, M. R. (2006, March). Modified pseudo LRU replacement algorithm. In 13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems (ECBS'06) (pp. 6-pp). IEEE.

[4] Robinson, J. T. (2004). Generalized Tree-LRU Replacement. Technical Report RC23332, IBM Research Division.

[5] Kim, S., Chandra, D., & Solihin, Y. (2004, October). Fair cache sharing and partitioning in a chip multiprocessor architecture. In Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004. (pp. 111-122). IEEE.

[6] Jaleel, A., Theobald, K. B., Steely Jr, S. C., & Emer, J. (2010). High performance cache replacement using re-reference interval prediction (RRIP). ACM SIGARCH computer architecture news, 38(3), 60-71.

[7] Qureshi, M. K., Jaleel, A., Patt, Y. N., Steely, S. C., & Emer, J. (2007). Adaptive insertion policies for high performance caching. ACM SIGARCH Computer Architecture News, 35(2), 381-391.

[8] Jaleel, A., Hasenplaugh, W., Qureshi, M., Sebot, J., Steely Jr, S., & Emer, J. (2008, October). Adaptive insertion policies for managing shared caches. In Proceedings of the 17th international conference on Parallel architectures and compilation techniques (pp. 208-219).

[9] Qureshi, M. K., & Patt, Y. N. (2006, December). Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06) (pp. 423-432). IEEE.

[10] Fang, J., Fan, Q., Hao, X., Cheng, Y., & Sun, L. (2017, May). Performance optimization by dynamically altering cache replacement algorithm in CPU-GPU heterogeneous multi-core architecture. In 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID) (pp. 723-726). IEEE.

[11] Moreto, M., Cazorla, F. J., Ramirez, A., & Valero, M. (2008). MLP-aware dynamic cache partitioning. In High Performance Embedded Architectures and Compilers: Third International Conference, HiPEAC 2008, Göteborg, Sweden, January 27-29, 2008. Proceedings 3 (pp. 337-352). Springer Berlin Heidelberg.

[12] Kumar, S., Zhao, H., Shriraman, A., Matthews, E., Dwarkadas, S., & Shannon, L. (2012, December). Amoeba-cache: Adaptive blocks for eliminating waste in the

memory hierarchy. In 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (pp. 376-388). IEEE.

[13] Kim, H., Lee, J., Lakshminarayana, N. B., Sim, J., Lim, J., & Pho, T. (2012). Macsim: A cpu-gpu heterogeneous simulation framework user guide. Georgia Institute of Technology.

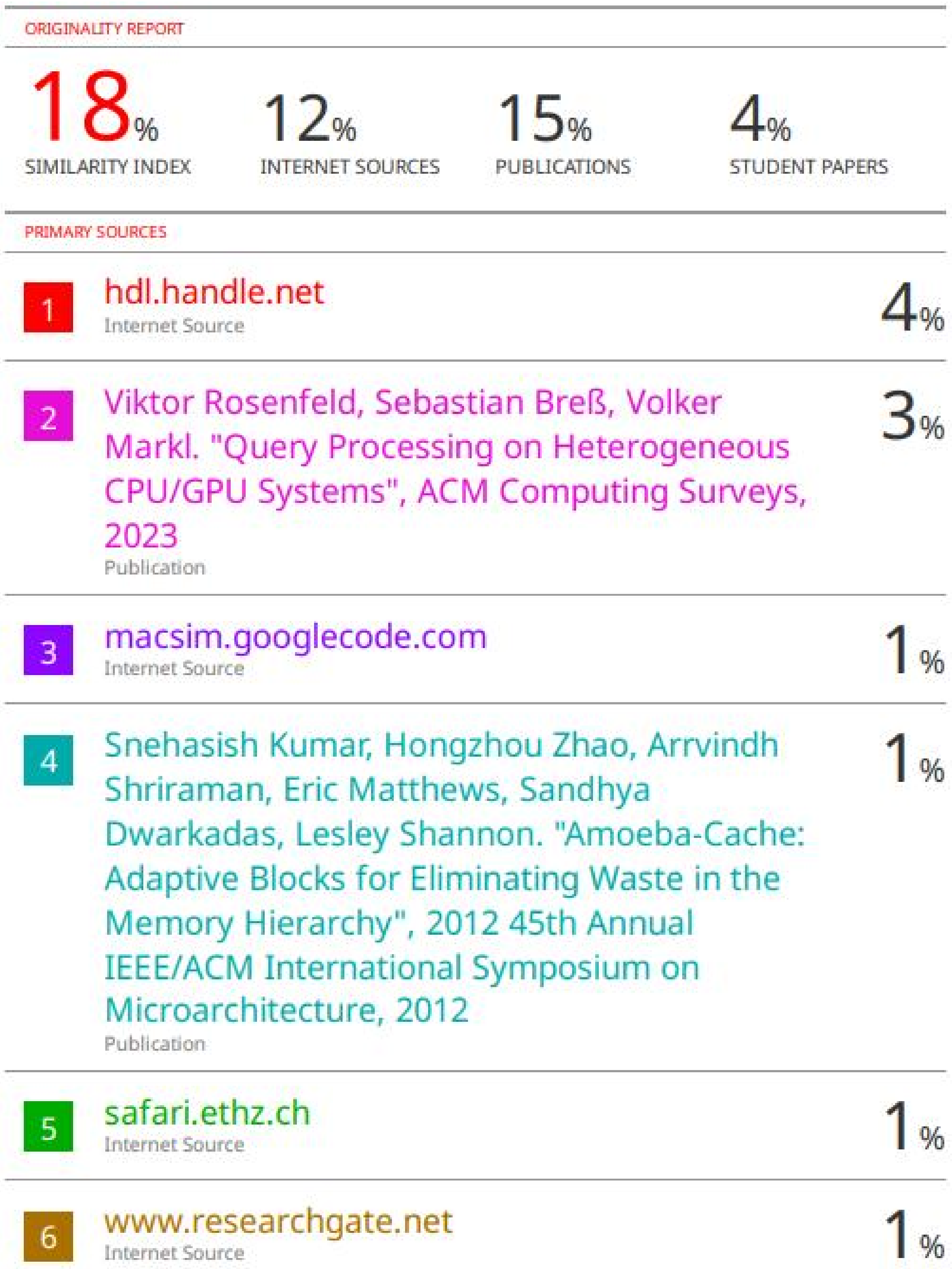
[14] Luk, C. K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., ... & Hazelwood, K. (2005). Pin: building customized program analysis tools with dynamic instrumentation. Acm sigplan notices, 40(6), 190-200.

[15] Li, B., Sun, J., Annavaram, M., & Kim, N. S. (2017, May). Elastic-cache: GPU cache architecture for efficient fine-and coarse-grained cache-line management. In 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS) (pp. 82-91). IEEE.

[16] Moreto, M., Cazorla, F. J., Ramirez, A., & Valero, M. (2011). Dynamic cache partitioning based on the MLP of cache misses. Transactions on High-Performance Embedded Architectures and Compilers III, 3-23.

[17] Kumar, S., Kalyanaraman, B., & Patil, S. EXTENDING CACHE REPLACEMENT POLICIES OF GPGPU-Sim (SRRIP, BRRIP, LFU).

[18] Rosenfeld, V., Breß, S., & Markl, V. (2022). Query processing on heterogeneous CPU/GPU systems. *ACM Computing Surveys (CSUR)*, *55*(1), 1-38.



ORIGINALITY REPORT

18%	12%	15%	4%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	hdl.handle.net Internet Source	4%
2	Viktor Rosenfeld, Sebastian Breß, Volker Markl. "Query Processing on Heterogeneous CPU/GPU Systems", ACM Computing Surveys, 2023 Publication	3%
3	macsim.googlecode.com Internet Source	1%
4	Snehasish Kumar, Hongzhou Zhao, Arrvindh Shriraman, Eric Matthews, Sandhya Dwarkadas, Lesley Shannon. "Amoeba-Cache: Adaptive Blocks for Eliminating Waste in the Memory Hierarchy", 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, 2012 Publication	1%
5	safari.ethz.ch Internet Source	1%
6	www.researchgate.net Internet Source	1%