SERBIAN JOURNAL OF ELECTRICAL ENGINEERING
Vol. 21, No. 2, June 2024, 217-234
UDC: 004.254:004.383.4 **DOI: https://doi.org/10.2298/SJEE2402217D**

An Analysis of Cache Configuration's Impacts on the Miss Rate of Big Data Applications Using Gem5

Hetal Vinubhai Dave[1], Nirali Ashwinbhai Kotak[2]

Abstract: This work aims to analyze the impacts of cache configurations on miss rates of big data benchmarks with varying level 1 instruction (L1I) and data (L1D) caches using the gem5 simulator. The cache miss rate of nine big data applications from four benchmark suits is analyzed with different cache configurations, such as increasing the cache size, varying the associativity, and altering the line size. The gem5 provides a versatile platform for conducting detailed experiments. The study sheds light on the relationship between cache and big data workloads, thus offering insights into optimizing cache configurations' effect on miss rates for improved performance.

Keywords: Cache configurations, Miss rate analysis, L1 instruction and data cache, Big data benchmark, Gem5.

1 Introduction

The performance of embedded processors is increased tremendously, prominent to the memory system needing the same speed. Unfortunately, the processor and the memory subsystem do not have the same rate, which creates a speed gap between the processor and the memory [1]. This gap will grow as CMOS technology reaches primary limits [2]. The cache is the fastest memory in the memory hierarchy. It drastically improves the memory access time from 60ns of DRAM to 10ns of L1 cache. However, the on-chip cache occupies a significant portion of the processor dies. It causes the cache to consume considerable processor power, i.e., 20% to 40% [3, 4].

Furthermore, an application's performance depends on the cache. So, the cache is an excellent candidate to be considered [4]. In today's world, almost all fields use comparatively large data, i.e., servers, social media [5], health [6, 7], business [7], IoT, and many more. With the wide use of big data applications, it is important to consider the cache for analysis. In tuning, a cache system designer has many parameters, including the cache size, associativity, and line size. Thus,

[1]Research Scholar, Gujarat Technological University, Ahmedabad, Gujarat, India;
E-mail: 209999915011@gtu.edu.in
[2]L. D. College of Engineering, Ahmedabad, Gujarat, India; E-mail: nakotak@ldce.ac.in

the designer must choose a suitable cache per system application to achieve a shorter time-to-market with a reasonably reduced design cost and effort [8]. Cache parameters influence the processor's performance to a considerable degree. Analyzing performance in terms of miss rate using various cache parameters is essential. Cache size, its associativity, and line size influence miss rate, so the processor is also affected. We verify these statements by running big data benchmark applications on the gem5. Benchmark suites facilitate designers to evaluate and compare designs. It is a set and representative of the applications program of interest. A benchmark application that is run on a simulator can mimic the characteristics of a particular real application system. Work carried out by [9] identified the characteristics of big data applications and their potential as big data applications. These applications are used in this work for simulation and analysis.

Moreover, the level 1 cache (on-chip cache) size is not kept too large. Thus, the processor can access information quickly because it needs to search a small area. The small storage space available can lead to an increased miss rate. Therefore, it is important to consider level 1 cache for an analysis. Furthermore, the importance of considering the L1I and L1D cache on the CPU performance is discussed in [10] in detail.

Thus, this work aims to analyze the L1I and L1D cache miss rate of nine big data applications from four benchmark suits using different cache configurations, such as increasing the cache size, varying the associativity, and altering the line size using the gem5. The cache configuration analysis for big data benchmarks includes a profound understanding of the different characteristics of these applications.

The findings of this study contribute to understanding cache impacts for big data applications.

Hence, it is summarized as:

– It examines cache configuration's impact on the miss rate for various big data benchmarks with varying L1I and L1D cache size, associativity, and line size.
– It shows the believability of the thumb rule [11] - the miss rate of the cache is directly mapped, and the cache size of N is equal to the miss rate of associativity is 2, and a cache size of N/2.

2 Big Data Benchmark

In embedded processor research, such as processor architecture and design, the designer often studies the response of embedded systems in detail. Thus,

designers need to examine how their new designs affect system performance and behavior at the early design time by running a benchmark application on a simulator that can mimic the characteristics of a particular real application. Scientific investigations in computer architecture research can be performed using the standard benchmarking technique. Standard big data benchmark applications are representative of text search, graph mining, machine learning, and statistics and mathematics categories and use different types of datasets like text, graph, image, and machine learning [9]. Benchmark applications and datasets are considered diverse to analyze cache miss rate. It covers the maximum possible domains of big data. Work carried out by [9] identified characteristics of big data applications, which are chosen for the simulation and analysis. These applications have been picked up through well-known C/C++-based standard benchmark suites, including Rodinia [12], Graphbig [13], Phoenix [14], and CortexSuite [15]. These benchmark applications may not be comprehensive, but these are wisely selected to be appropriately representative of our study.

LR (Linear Regression) and SM (String Match) are the text-processing applications. These applications process large text files as input. An LR can be categorized as a statistical and mathematical domain, and it creates the summary measurements of points to give the linear approximation of all the points. An SM can be classified as a text search domain, and it scrolls through a list of keys to determine if any of them occur in a list of encrypted words of text input files. Applications like BFS (Breadth-First Search), CC (Connected Component), DFS (Depth-First Search), and PR (Page Rank) are classified as graph processing domains and work on large input graph datasets having a list of edges and vertices. A BFS is a prevalent and widely used graph-processing application [16]. It generates the graph by specifying the number of nodes. A CC computes the total number of vertices and edges and connected components of the given graph. A DFS performs graph traversal operations while the PR assigns a rank to each node and measures the graph node's connectivity. A Histogram application processes a large image file as input data and falls under the statistical and mathematical domain. It gives the frequency of occurrence for an image file's RGB (Red, Green, Blue). PCA (Principle Component Analysis) and K-means are machine learning domain applications. A PCA extracts features from multivariate image datasets and analyses chosen correlations. A K-means works on a dataset that contains various numeric features. A detailed explanation of each big data benchmark application is specified in **Table 1**. All big data benchmark applications were compiled with the help of a make-file provided with individual applications to convert their source code to binary or executable code.

Table 1:
Big data benchmarks with their description.

Domain	Application name	Benchmark suite	Description	Data type
Text Search	SM (String Match)	Phoenix 2.0 [14]	Scans a list of keys to decide if any of them occur in a list of encrypted words [9]	A text file contains a list of encrypted words
Graph Processing	BFS (Breadth-First Search)	Rodinia [12]	Large graphs involving millions of vertices.	Graph generated by specifying the number of nodes
	CC (Connected Component)	GraphBig [13]	Computes a set of connected subgraphs from the given graph	Graph represented in terms of vertex and edge file
	DFS(Depth-First Search)	GraphBig [13]	Graph traversal operations	Graph represented in terms of vertex and edge
	Page rank	GraphBig [13]	Assign rank to each node and measure the graph nodes' connectivity	Graph represented in terms of vertex and edge
Statistical and mathematical	Histogram	Phoenix 2.0 [14]	Computes the RGB histogram of an image and determines the frequency of each RGB component in a set of images	Image file
	LR(Linear Regression)	Phoenix 2.0 [14]	Generates the summary measurements of points to give the linear approximation of all the points.	Text file
Machine learning	PCA (Principle Component Analysis)	Cortex Suite [15]	Extracts features from Multivariate dataset.	Image data in numeric form
	K-means	Rodinia [12]	Indicates the data by centroids of the sub-clusters by dividing a data cluster into K sub-clusters [9].	The dataset with numeric structures.

3 Cache Memory

Recently used instructions and data are stored in small, fast memory [2]. The processor first checks to see if the data it needs to process is already in the cache. If it is, it operates on this data. Cache configuration is an essential part of embedded system design. An oversized cache leads to an extra silicon area and more power consumption. Further, the undersized cache leads to degraded performance.

Moreover, different applications need different cache configurations [8], i.e., some require a small cache, while others need a large size. Some are related to associativity, while others are not. Some require a larger line size, while others require a smaller one. Therefore, the best parameters will vary by type of application. Thus, the system designer must carefully choose cache configuration. A careful analysis is essential at this level to make reliable architectural decisions.

The cache configuration and its effect are explained below.

The cache design constraints are cache size, associativity, line size, replacement policy (FIFO, LRU, and Random), writing policy (write back and write through), cache type, and cache level. The possible combination of design parameters is called the design point. The design points are considered as the design space altogether. To find the best design configuration from the obtainable design choices, a compelling design space exploration (DSE) process is essential [8]. The cache configuration strongly affects the performance. Therefore, analyzing the cache is important in choosing the best cache through the DSE technique.

Cache size is measured capacity in bytes. Generally, size is in powers of 2, 4, 8, 16, etc., and the unit is KB up to MB. The more significant cache can store more data closer to the processor. Therefore, different processors have caches of different sizes. Associativity means how data from the main memory could be mapped onto the cache using different mapping techniques [12].There are three types of it: 1) Direct mapped, 2) Full associative, and 3) N-way set associative [18]. In the direct map (one-way), a single data location of the main memory is mapped with a single cache location. In full associative, any data location of the main memory is mapped with any place of cache. In the n-way set associative, the cache is designed using the n-set. Line size represents the amount of data read or written in each cache access [8]. A line refers to data transferred between the cache and main memory. A larger line size holds more words in the cache. The commonly used line size is 16 to 256 bytes. The application can influence the line size's usefulness that the system designer chooses. Line size is also called block size. There are two types of cache. 1) Instruction and data cache 2) Unified and split cache. An instruction cache is known as an I-cache, which stores instructions only. The data cache is known as D-cache, which stores data only. A unified cache stores instructions and data, while a split cache consists of independent units – an I-cache and a D-cache [18]. The cache is designed with different levels.

These are called Level 1(L1), Level 2(L2), and Level (L3). L1 cache is designed in the same chip area. L1 is also called on-chip memory. Due to on-chip L1, the processor tends to have less access to the external bus. The L2 and L3 are off-chip, bigger, slower, and less expensive than the L1 cache. The cache is explained in detail by [11, 19].

Each cache configuration parameter affects its performance parameters: hit rate, miss rate, power consumption, access time, and hardware complexity to a varying degree. An analysis of these parameters helps assess the impact of cache on system performance. The processor checks the cache first to see if it needs any information, i.e., data or instruction. It is called a cache hit if information is accessible in the cache. It is called a cache miss if the information is unavailable in the cache. If a miss occurs, the missed line is fetched from the main memory, which is time-consuming. The processor uses a fetched line and saves it to the cache [18]. Three types of misses exist: 1) compulsory miss occurs in the memory location accessed the first time, 2) capacity misses occur due to small space, and 3) conflict misses occur due to insufficient space when two lines are mapped on the exact location. The miss rate is defined as how often the cache's desired memory location is absent. A larger cache size decreases the miss rate.

Furthermore, hardware complexity, cost, conflict miss, and power consumption will increase if the cache size is more prominent. It slows down access time. An associativity is also essential in the effect of cache performance parameters. Direct mapped cache increases the miss rate. Further, it decreases the hit rate and cost. However, it supports the fastest access time [8]. The set associative cache decreases the miss rate. Intended for a specific cache size, associativity can lessen the miss rate [20], i.e., Let's intend the cache size is 8MB, and associativity is changed from one way to two ways, then this associativity cache retains almost 44% of the cache misses [21]. Larger caches with a higher set associativity have lower miss rates but longer access times. A full associative cache supports the lowest miss rate and conflict misses, but its internal design is very complex, and it consumes the highest power. Therefore, the correct associative value for each cache is important; hence, simulation is carried out using different associativity in this work. Line size is essential when designing a processor’s cache. The designer cannot keep a line size too small or too large as both have pros and cons; a larger line size decreases the miss rate and compulsory misses. Further, it increases conflict misses. Moreover, it supports fast access time.

Locality of reference is a working principle of cache. It supports temporal locality, which means the same data is requested again very soon, and spatial locality means location is referenced, and its data is brought into the cache; its nearby location’s data is also obtainable in the future.

An analysis of cache configuration helps understand its impact on system performance, identify possible limitations, and improve overall system efficiency.

4 About the Gem5 Simulator

The gem5 [22, 23] is a popular open-source simulator for the computer architecture research community. The gem5 is well-maintained by developers and broadly considered in academia, research, and industry. It combines the M5 simulator [24] and GEMS(General Execution Driven Multiprocessor Simulator) known as gem5. The gem5 is written primarily in C++ and Python. It can run interactive benchmarks. The gem5 allows the construction of several aspects of the simulated system, including CPU models, memory hierarchy, and cache configurations. It supports great flexibility in configuring various system parameters. The gem5 aids many instruction set architectures (ISA), i.e., ARM, X86, MIPS, POWER, RISC-V, ALPHA, and SPARC. The gem5 includes various CPU types, i.e., Atomic-Simple, Timing-Simple, Out-of-Order (O3), In-Order, and KVM. The simulator provides two cache memory models, i.e., classic cache and ruby cache. It supports full system (FS) and syscall emulation (SE) simulation mode. In the FS mode, the entire operating system is simulated. In contrast, in SE mode simulation, the gem5 executes user-mode binaries without executing the kernel-mode system calls of a real operating system [23].

After simulation, as an output, three files are produced for each run in the m5out folder by the gem5, named config.ini, config.json, and stats.txt. The generated output files show many parameters. gem5 version 22.1 is used in this work, which was released in the year 2022. X86 ISA, Timing-Simple CPU, SE simulation mode, and classic cache memory model from the gem5 are used to carry out this work. The simulation result is analyzed using a stats.txt output file, which provides valuable statistics. Comparison is done on various cache parameters like cache size, associativity, and line size of L1I and L1D for the mentioned benchmark applications with its input data to analyze how changes in them might affect the overall cache performance in terms of miss rate.

Some other computer architecture simulators are SimOS, Simics, MARSSx8, Simplescalar, and many more are explained in detail by [25].

5 Related Work

Prior efforts have explored cache configuration, with some studies focusing on specific cache parameters, simulation tools, or different benchmark applications.

The authors [26] investigated the L1I and L1D cache line utilization of mobile workloads for false sharing using the gem5. They found an effect of false sharing on the miss rate by varying cache line size from 16 to 256 bytes. The authors [27] characterized SPEC CPU2017 benchmark applications by examining cache, instruction mix, and execution time. They use hardware performance counters on a real system. They found L1, L2, and L3 cache miss rates, which are essential for the application's overall performance. As per their

experiment, L3 has a lower miss rate than L2 for CPU2017 applications. The authors [28] estimated a cache hit rate via their proposed model for embedded systems. They used three mibench benchmarks to estimate their model. The authors [29] evaluate the performance of twenty applications from the SPEC CPU2006 benchmark suite in chip multiprocessors by taking IPC and CPI as matric with varying cache sizes. The authors [10] explore the variations in performance affected by altering the L1I and L1D cache size. They run a FreqMine application from the PARSEC suite using gem5. They have chosen various parameters to analyze, i.e., miss rate, instruction rate, memory latency, and bus traffic. The authors [30] analyzed the STT-RAM cache energy and latency of wearable workloads applications of cache configurations such as increasing the cache size from 4 to 32 KB, varying associativity from 2 to 16-way, and altering line size from 16 to 64 bytes using the gem5. The authors [31] analyzed the miss rate and IPC of mibench benchmark applications using the gem5 by considering L1 cache size and latency. They concluded that the system performance improved as the latency decreased and the L1 cache size increased. The authors [32] analyzed memory bandwidth, system bus output, and cache miss rate of SPLASH-2 benchmarks using gem5 on ARM and ALPHA ISA. They observed that memory bandwidth, system bus output, and miss rate decreased as cache size increased. The authors [33] run Bernstein's attacks, which are cryptographic operations on the gem5, to analyze cache configurations for performance measurement. The authors [34] identified the L1D cache sensitivity of three applications, libquantum, h264ref, and hmmer, from the SPEC CPU 2006 benchmark for big data benchmarking. The authors [35] analyzed high-performance graph algorithms. They experimented using hardware performance counters on an Ivy Bridge Server. Representative graph applications are Breadth-First Search, Single-Source Shortest Paths, PageRank, Connected Components, and Betweenness Centrality. They concluded by saying that there is no perfect solution for performance tests of graph algorithms.

An analysis of graph processing is important for big data applications, [36] authors analyzed graph processing applications to discover the multi-core and cache bottleneck. They used SNIPER and CACTI tools. Graph applications are Breadth-First Search, Single-Source Shortest Paths, PageRank, Connected Components, and Betweenness Centrality from the GAP benchmark used in their work. They concluded that the L2 cache contributes less performance while the L3 cache is sensitive towards higher performance for graph applications. The authors [37] characterized big data applications on big-Xeon and little-Atom-based Core server architectures. They measured power and performance at the system and architecture level. They revealed the effect of application type, data size, and performance limits on selecting servers and handling big data. Big data applications include graph processing, data mining, data analysis platforms, and pattern searching. They provide cache-related insight for both servers. A small

two-level data cache is adequate for a little core server, and improvement is needed for instruction cache pipeline design for big data processing. Big data applications affect L1 cache miss rate and branch predictor accuracy in big and little core servers. They considered seven cache parameters: L1 cache size and its associativity, L2 cache size and its associativity, cache line size, replacement policy, and cache clusivity. The authors [16] estimated the performance of a mix of graph processing- GAP benchmark, scientific-XSBench benchmark, and industrial- Qualcomm application workloads along with the SPEC CPU 2006 and SPEC CPU 2017 workloads. They experimented using ChampSim with last-level cache replacement policies.

The prior studied work is related to traditional benchmarks, gem5, and big data applications analysis for cache configuration, but a limited focus on new-fangled effort going on big data applications. Most of the previous research has explored cache analysis by considering SPEC CPU and mibench benchmark workloads. In this work, considered applications are representative of text search, graph mining, machine learning, and statistics and mathematics categories and use different types of datasets like text, graph, image, and machine learning. This study aims to connect this gap and contribute insights specific to this area by examining cache configuration's impact on the miss rate of various big data benchmarks with varying L1I and L1D cache size, associativity, and line size.

6 Methodology and Simulation Process

This section details the methodology and simulation process used to experiment. We used gem5 with different cache configurations. The cache design has a substantial effect on application performance. Cache sizes from 1KB to 1024KB, associativity from 1 to 16 ways, and line sizes from 8 to 64 bytes are considered for L1I and L1D caches. The default replacement policy on gem5 is LRU for the cache configuration. Each simulation output was usually redirected to a static.txt file as an output of the gem5.

First, all the prerequisites for gem5 are installed on the Ubuntu host machine. Then, gem5 was cloned and built. The gem5 was built for X86 architectures using the build command. Then, we collected, cloned, and learned various big data applications. All big data applications were compiled using the make file provided with each application. The make file uses GCC and G++ to convert the source code of an application written in C and C++ into binary or executable code on the host system Ubuntu version 20.04. Generated binary or executable code with its input dataset file was simulated on the gem5. A compiled benchmark application binary code must be passed to gem5 using command-line options for the simulation on gem5. The gem5 can read the configuration script from the command line. Therefore, the configuration script was prepared for the gem5.

The prepared configuration script includes gem5 built option that is X86 ISA, SE mode, CPU type, i.e., Timing-Simple, benchmark application binary code with its input file, and cache-related parameters like L1I and L1D cache size, its associativity, and line size.

The cache parameters are the cache size, associativity, and line size for the gem5 commands. Each benchmark application program was run using different cache parameters as below: 1) Cache size: 1KB to 1024KB, 2) Associativity: 1-way to 16-way, 3) Line size: 8 to 64 bytes, and 4) Type: instruction and data level 1. All sizes are the power of two. Each benchmark application was simulated using gem5 with varying L1I and L1D cache size, associativity, and line size. Upon the completion of each simulation, the stats.txt file is considered to collect the miss rate and plot the graph for analysis. As mentioned before, standard big data applications are representative of text search, graph processing, machine learning, and statistics and mathematics categories[9] and use different types of datasets like text, graph, image, and machine learning. These applications are from well-known C/C++-based standard benchmark suites including Rodinia [12], Graphbig [13], Phoenix [14], and CortexSuite [15].

Simulation is done on a host machine with i5-4300M CPU, 8.00 GB RAM, and a 64-bit Ubuntu 20.04 (Linux) operating system.

7 Simulation Result and Observation

This section presents experiment results and observations derived from the analysis of miss rates of big data applications, which are SM, BFS, CC, DFS, PR, Histogram, LR, PCA, and K-means. The experimental analysis is carried out with various cache configurations like the cache size, associativity, and line size of L1I and L1D using the gem5.

Fig. 1 shows the outcome of the miss rate by varying L1I cache size from 1KB to 1024KB with associativity one and line size 64 bytes for different big data applications. The highest and lowest L1I miss rate is reported for CC and histogram application, respectively, for 1KB L1I size. If the program code is large and the cache size is small, the cache cannot store the entire program code, leading to a greater miss rate in many modern interactive applications. Thus, it is shown in Fig. 1 that CC, DFS, PR, and BFS significantly use the L1I cache of more than 64KB. Here, it is observed that those applications that have higher program code sizes face a high miss rate for small L1I sizes, i.e., CC, DFS, PR, BFS, and K-means compared to the applications that have smaller program code sizes, i.e., histogram, SM, LR, and PCA. Higher than 16KB L1I size provides a constant miss-rate for applications like histogram, LR, SM, and K-means. The miss rate for PCA has decreased from 1KB to 128KB.

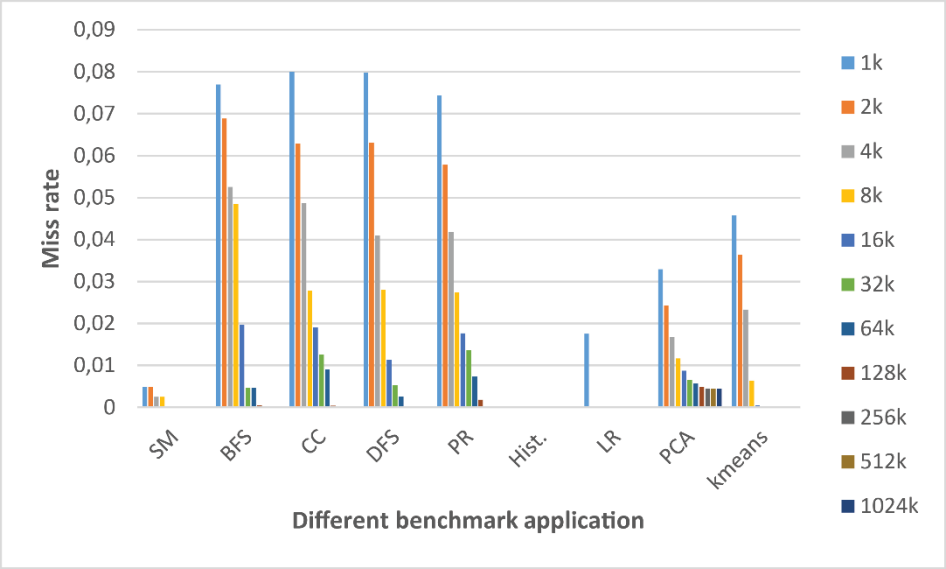

Fig. 1 – *An effect of varying direct-mapped L1I cache size on miss rate.*

Fig. 2 reveals the miss rate of the L1D cache when the associativity is one, the line size is 64 bytes, and the cache size varies from 1KB to 1024KB for different big data applications.

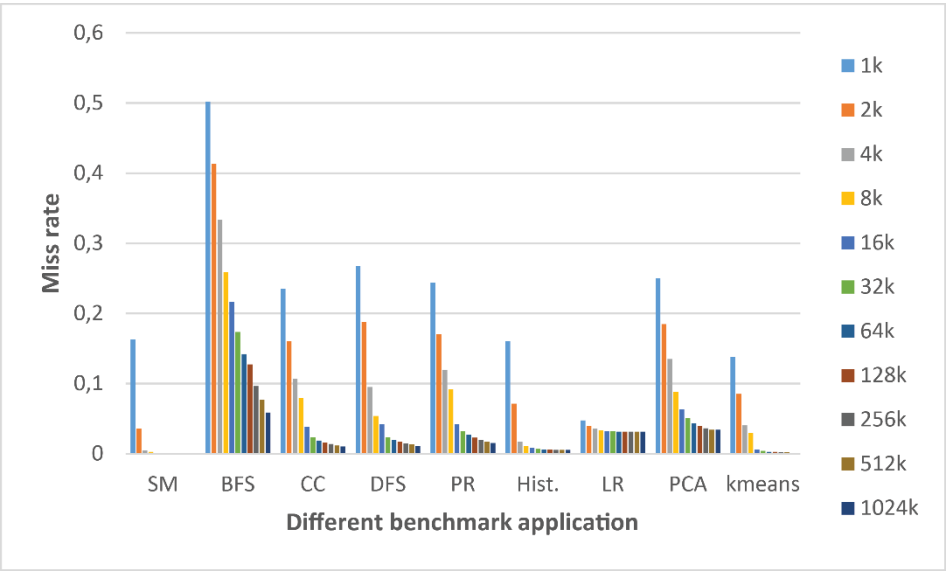

Fig. 2 – *An effect of varying direct-L1D cache size on miss rate.*

Diverse input data sets like graphs, images, and text are considered to carry out the simulations. The graph processing applications are BFS, CC, DFS, and PR. Histogram and PCA process an image input file. LR and SM are text-processing applications. As Fig. 2 exhibits, the highest miss rate is stated for BFS. BFS is a graph processing application that passes through all vertices in the data set. Each iteration will access or modify other vertices in the data. Since these other vertices are not certainly consecutive, this creates an irregular access pattern that is possibly hard to predict. Therefore, a high L1D miss is made. It is noted that increased L1D cache size from 1KB to 1024KB is beneficial as the miss rate is decreased for the graph processing applications. The miss rate is reduced from 1KB to 256KB for L1D size. Larger sizes beyond 256KB only provide minimal benefits for image processing applications. For text processing applications, the reported miss rate is not much benefited by larger L1D sizes.

Another observation is the overall miss rate of L1I is lower than L1D of all the applications, i.e., with 1KB L1I cache size, the highest miss rate is 0.08 for the CC application, and 1KB L1D cache size, the highest miss rate is 0.5 for BFS application. The reason is that instruction also leans towards a better locality of reference, both spatial and temporal, causing a lower miss rate for the instruction cache. Hence, there is a considerable chance to improve the processor's performance for big data applications through a larger L1D cache.

L1I cache size is 32KB, line size is 64 bytes, and associativity varying from 1 to 16 is chosen and calculated. The miss rate of big data applications is presented in Fig. 3. It is shown here that the direct-mapped cache gives the highest miss rate.

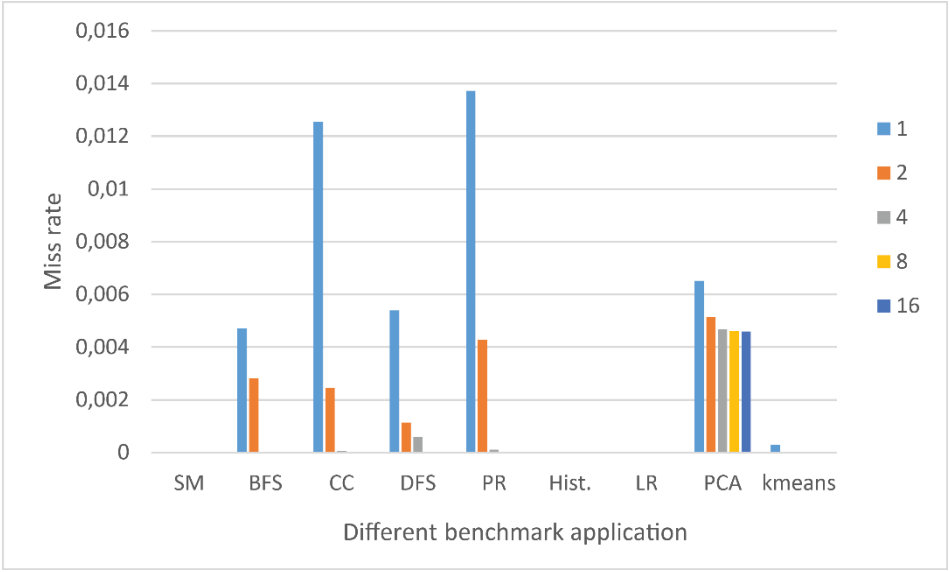

Fig. 3 – *An effect of varying L1I cache associativity on miss rate.*

Higher associativity means more lines in one set, which offers fewer chances of data not being available in the cache, thus reducing the miss rate. The higher associativity gives better performance, but it increases the cost and cache management complexity. As associativity is increased from one to four, the miss rate is decreased for applications like BFS, CC, DFS, PR, PCA, and K-means.

The observation is that most of the applications with caches implemented with either two- or four-way associativity give a lower miss rate, as increasing the associativity further than this is shown to have less effect on the miss rate. Increasing associativity does not give any changes in the change of the miss rate for histogram, LR, and SM applications.

Fig. 4 represents the miss rate achieved from the simulation of an L1D cache size of 32KB, line size of 64 bytes, and associativity varying from 1 to 16 for big data applications. The miss rate is decreased at associativity four for most of the applications. There is no significant benefit in the miss rate achieved for associativity eight and sixteen way.

The rule of thumb for cache says the miss rate of cache is directly mapped, and the cache size of N is equal to the miss rate of associativity is two, and a cache size of N/2 [11]. This is valid for applications like PCA, PR, and LR.

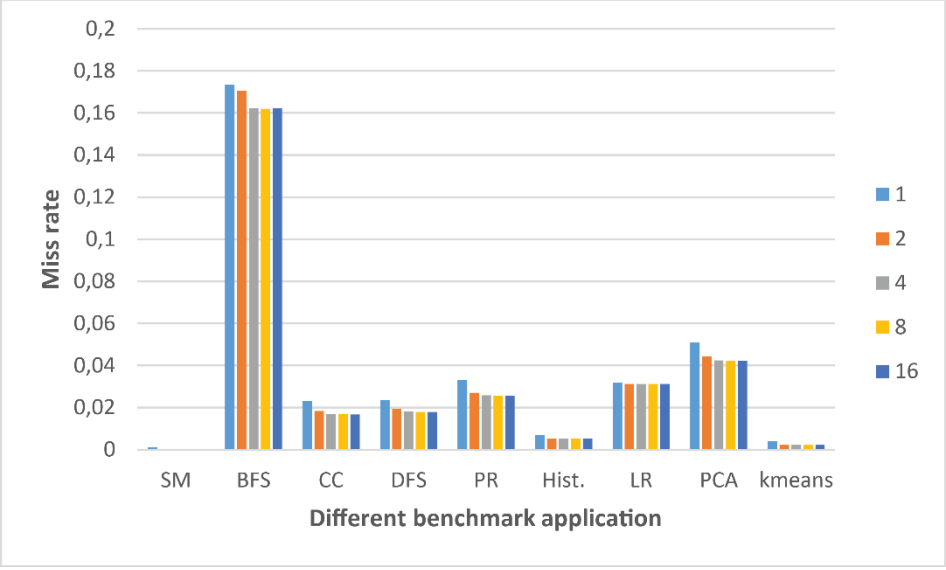

Fig. 4 – *An effect of varying L1D cache associativity on miss rate.*

The miss rate of big data applications for the L1I cache size is 32 KB, associativity is one, and varying line size from 8 to 64 bytes is given in Fig. 5. The highest and lowest miss rate is reported for PR and SM, respectively. It is noted that there is no significant effect on the miss rate by changing the L1I line

size from 16 to 64 bytes for histogram, LR, SM, and K-means applications. The miss rate decreased as line size increased from 8 to 64 bytes for applications like BFS, CC, DFS, PR, and PCA.

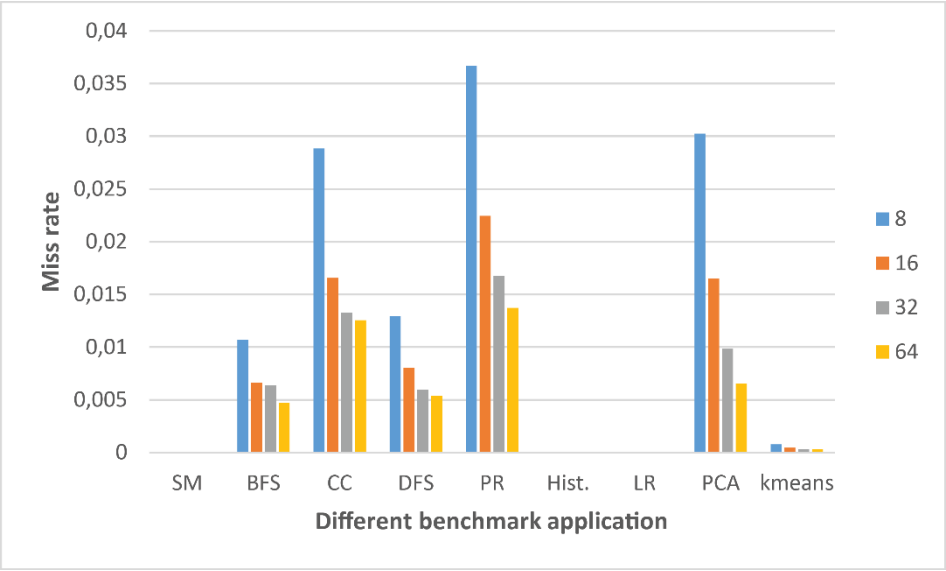

Fig. 5 – *An effect of varying direct-mapped L1 cache line size on miss rate.*

The simulation result is presented in Fig. 6 for the L1D cache size is 32KB, associativity is one, and varying line size from 8 to 64 bytes.

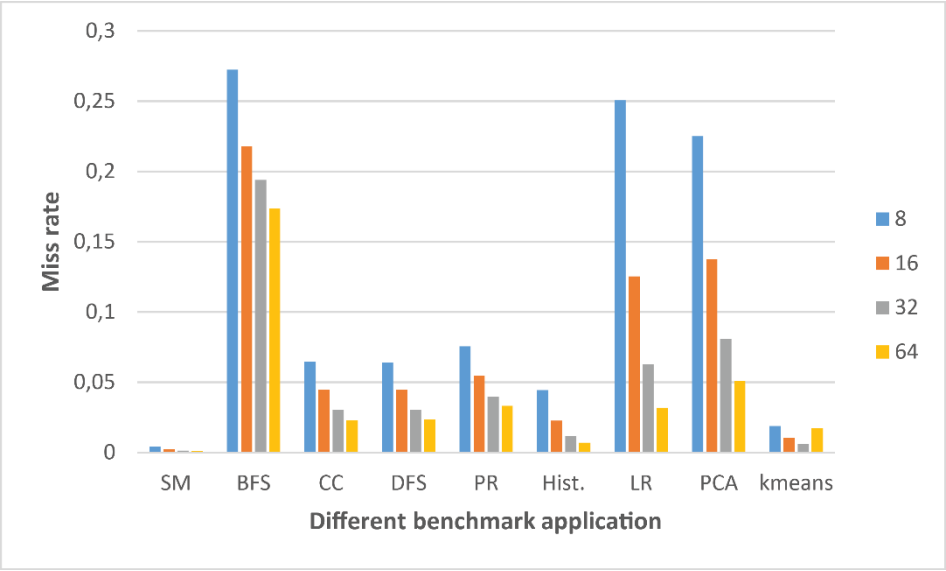

Fig. 6 – *An effect of varying direct-mapped L1D cache line size on miss rate.*

A larger line size supports spatial locality, which means location is referenced, and its data are brought into the cache; its nearby location data are also obtainable in the future. Thus, as line size increases, all applications' miss rate decreases. Hence, a 64-byte L1D line size is beneficial for achieving a reasonable miss rate.

8 Conclusion

In this work, we have considered the impact of cache configurations on the miss rate of big data benchmarks using gem5. The miss rate is obtained by varying parameters such as cache size, associativity, and line size of L1I and L1D cache configuration. An analysis of cache parameters is important in choosing the best cache, and an application's performance depends on the cache. From our simulations and resulting graphs plotted, it is concluded that the applications face a high miss rate for small L1I sizes that have higher program code sizes. The cache size increases from 1KB to 128KB, and the miss rate decreases. Therefore, keeping the L1I cache size much larger is not always good. An increased L1D cache size from 1KB to 1024KB is beneficial as the miss rate is decreased for the graph processing applications. The L1D size beyond 256KB only provides minimal benefits for image processing applications. The miss rate is not much benefited by larger L1D size for text processing applications. The overall miss rate of L1I is lower than L1D of all the applications. The higher associativity performs better but increases the cost and cache management complexity. The rule of thumb for the cache is valid for applications like PCA, PR, and LR. The 64-byte L1D line size is beneficial for achieving a reasonable miss rate for all the applications. This analysis will be helpful for further understanding the effect of cache configurations' impact on the miss rate of big data applications when selecting the suitable cache configuration through DSE. Thus helping optimize performance for the given application in a fair amount of time.

This work examines the Level 1(L1) cache for big data applications. In the future, different levels of caches can be examined, i.e., Level 2(L2) and Level 3(L3) caches.

This study obtained the cache miss rate as a performance parameter from simulation results, and an analysis was carried out. The on-chip cache occupies a significant portion of the processor dies. It causes, it consumes a considerable amount of processor power. Thus, in the future, cache analysis will consider power as a performance parameter, vital to studying power-efficient cache configuration using gem5. Moreover, other cache performance parameters like IPC (instructions per cycle) and execution time can be analyzed for big data applications using the gem5 in the future.

Our study analyzes cache configurations' impact on the miss rate of big data applications for X86 ISA. Different ISAs from gem5, like ARM, MIPS, ALPHA, and RISC-V, can be considered in the future.

9 References

[1] O. Navarro, J. Yudi, J. Hoffmann, H. G. Muñoz Hernandez, M. Hübner: A Machine Learning Methodology for Cache Memory Design Based on Dynamic Instructions, ACM Transactions on Embedded Computing Systems, Vol. 19, No. 2, March 2020, p. 12.

[2] J. L. Hennessy, D. A. Patterson: A New Golden Age for Computer Architecture, Communication of the ACM, Vol. 62, No. 2, January 2019, pp. 48 – 60.

[3] J. Díaz Álvarez, J. M. Colmenar, J. L. Risco-Martín, J. Lanchares, O. Garnica: Optimizing L1 Cache for Embedded Systems Through Grammatical Evolution, Soft Computing, Vol. 20, No. 6, June 2016, pp. 2451 – 2465.

[4] R. Vazquez, A. Gordon-Ross, G. Stitt: Energy Prediction for Cache Tuning in Embedded Systems, Proceedings of the IEEE 37th International Conference on Computer Design (ICCD), Abu Dhabi, United Arab Emirates, November 2019, pp. 630 – 637.

[5] S. Rahman, H. Reza: A Systematic Review Towards Big Data Analytics in Social Media, Big Data Mining and Analytics, Vol. 5, No. 3, September 2022, pp. 228 – 244.

[6] W. Raghupathi, V. Raghupathi, V. Raghupathi: Big Data Analytics in Healthcare: Promise and Potential, Health Information Science and Systems, Vol. 2, February 2014, p. 3.

[7] E. K. Lee: Innovation in Big Data Analytics: Applications of Mathematical Programming in Medicine and Healthcare, Proceedings of the IEEE International Conference on Big Data, Boston, USA, December 2017, pp. 3586 – 3595.

[8] K. Balasubadra, A. P. Shanthi, V. P. Srinivasan: Hybrid Design Space Exploration Methodology for Application-Specific System Design, International Journal of New Computer Architectures and Their Applications, Vol. 7, No. 3, July 2017, pp. 102 – 111.

[9] H. Ahmed, M. Ali Ismail: Towards a Novel Framework for Automatic Big Data Detection, IEEE Access, Vol. 8, October 2020, pp. 186304 – 186322.

[10] R. Akula, K. Jain, D. J. Kotecha: System Performance with Varying L1 Instruction and Data Cache Sizes: An Empirical Analysis, arXiv:1911.11642 [cs.PF], November 2019, pp. 1 – 16.

[11] J. L. Hennessy, D. A. Patterson: Computer Architecture: A Quantitative Approach, 4th Edition, Morgan Kaufmann, Amsterdam, Boston, 2007.

[12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.- H. Lee, K. Skadron: Rodinia: A Benchmark Suite for Heterogeneous Computing, Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), Austin, USA, October 2009, pp. 44 – 54.

[13] L. Nai, Y. Xia, I. G. Tanase, H. Kim, C.- Y. Lin: GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions, Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Austin, USA, November 2015, pp. 1 – 12.

[14] R. M. Yoo, A. Romano, C. Kozyrakis: Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System, Proceedings of the IEEE International Symposium on Workload Characterizationn (IISWC), Austin, USA, October 2009, pp. 198 – 207.

[15] S. Thomas, C. Gohkale, E. Tanuwidjaja, T. Chong, D. Lau, S. Garcia, M. Bedford Taylor: CortexSuite: A Synthetic Brain Benchmark Suite, Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), Raleigh, USA, October 2014, pp. 76 – 79.

[16] A. V. Jamet, L. Alvarez, D. A. Jiménez, M. Casas: Characterizing the Impact of Last-Level Cache Replacement Policies on Big-Data Workloads, Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), Beijing, China, October 2020, pp. 134 – 144.

[17] M. Gupta, J. Singh: A Comparative Study of Cache Optimization Techniques and Cache Mapping Techniques, International Journal of Engineering Research and Technology, Vol. 6, No. 5, May 2017, pp. 903 – 905.

[18] A. O. Abayomi, A. A. Olukayode, G. O. Olakunle: An Overview of Cache Memory in Memory Management, Automation, Control and Intelligent Systems, Vol. 8, No. 3, October 2020, pp. 24 – 28.

[19] G. Jain: Memory Models for Embedded Multicore Architecture, Ch. 4, Real World Multicore Embedded Systems, 1st Edition, Elsevier Inc., Amsterdam, Boston, 2013.

[20] T. I. O. Ahmed, E. M. Elamin: Design Strategy of Cache Memory for Computer Performance Improvement, International Journal of Research Studies in Electrical and Electronics Engineering, Vol. 4, No. 3, 2018, pp. 12 – 17.

[21] J. L. Hennessy, D. A. Patterson: In Praise of Computer Organization and Design: The Hardware / Software Interface, 5th Edition, Morgan Kaufmann, Amsterdam, Boston, 2014.

[22] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, D. A. Wood: The gem5 Simulator, ACM SIGARCH Computer Architecture News, Vol. 39, No. 2, August 2011, pp. 1 – 7.

[23] J. Lowe-Power et al.: The gem5 Simulator: Version 20.0+*, arXiv:2007.03152 [cs.AR], July 2020, pp. 1 – 21.

[24] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, S. K. Reinhardt: The M5 Simulator: Modeling Networked Systems, IEEE Micro, Vol. 26, No. 4, July-August 2006, pp. 52 – 60.

[25] A. Akram, L. Sawalha: A Survey of Computer Architecture Simulation Techniques and Tools, IEEE Access, Vol. 7, May 2019, pp. 78120 – 78145.

[26] A. Van Laer, W. Wang, C. Emmons: Inefficiencies in the Cache Hierarchy: A Sensitivity Study of Cacheline Size with Mobile Workloads, Proceedings of the International Symposium on Memory Systems (MEMSYS), Washington, USA, October 2015, pp. 235 – 245.

[27] A. Limaye, T. Adegbija: A Workload Characterization of the SPEC CPU2017 Benchmark Suite, IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Belfast, UK, April 2018, pp. 149 – 158.

[28] V. C. Chijindu, O. K. Ugwueze, C. C. Udeze, M. A. Ahaneku, J. N. Eneh, O. M. Ezeja, E. C. Anoliefo: Modeling Cache Performance for Embedded Systems, Bulletin of Electrical Engineering and Informatics, Vol. 10, No. 5, October 2021, pp. 2910 – 2920.

[29] P. Jain, S. K. Surve: Evaluating Resource Centric Behavior of Workloads and Performance Analysis in CMPs due to Shared Resources, International Journal of Engineering and Advanced Technology, Vol. 8, No. 6, August 2019, pp. 4974 – 4981.

[30] D. Gajaria, T. Adegbija: Evaluating the Performance and Energy of STT-RAM Caches for Real-World Wearable Workloads, Future Generation Computer Systems, Vol. 136, November 2022, pp. 231 – 240.

[31] R. Saha, Y. P. Pundir, S. Yadav, P. K. Pal: Impact of Size, Latency of Cache-L1 and Workload Over System Performance, Proceedings of the International Conference on Advances in

Computing, Communication and Materials (ICACCM), Dehradun, India, August 2020, pp. 390 – 393.

[32] B. Vikas, B. Talawar: On the Cache Behavior of SPLASH-2 Benchmarks on ARM and ALPHA Processors in Gem5 Full System Simulator, Proceedings of the 3rd International Conference on Eco-friendly Computing & Communication Systems, Mangalore, India, December 2014, pp. 5 – 8.

[33] X. Yu, Y. Xiao, K. W. Cameron, D. Yao: Comparative Measurement of Cache Configurations' Impacts on Cache Timing Side-Channel Attacks, Proceedings of the 12th USENIX Workshop on Cyber Security Experimentation and Test (CSET), Santa Clara, USA, August 2019, pp. 1 – 9.

[34] K. Hurt, E. John: Analysis of Memory Sensitive SPEC CPU2006 Integer Benchmarks for Big Data Benchmarking, Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems, Austin, USA, February 2015, pp. 11 – 16.

[35] S. Beamer, K. Asanovic, D. Patterson: Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server, Proceedings of the IEEE International Symposium on Workload Characterization, Atlanta, USA, October 2015, pp. 56 – 65.

[36] A. Basak, X. Hu, S. Li, S. M. Oh, Y. Xie: Exploring Core and Cache Hierarchy Bottlenecks in Graph Processing Workloads, IEEE Computer Architecture Letters, Vol. 17, No. 2, July-December 2018, pp. 197 – 200.

[37] M. Malik, S. Rafatirad, H. Homayoun: System and Architecture Level Characterization of Big Data Applications on Big and Little Core Server Architectures, ACM Transactions on Modeling and Performance Evaluation of Computing Systems, Vol. 3, No. 3, July 2018, p. 14.