Evaluation of the Impact of Coherence Protocols and Cache Sizes on Parallel Algorithms Through Simulations

Guilherme Dantas C. Fagundes, Matheus Alcântara Souza

Depto. de Ciência da Computação – Pontifícia Universidade Católica de Minas Gerais
Belo Horizonte – MG – Brasil

gdcfagundes@sga.pucminas.br, matheusalcantara@pucminas.br

Abstract. *This article explores the intersection between parallel algorithms and cache optimization, focusing on how different coherence protocols and cache sizes impact the performance of parallel algorithms. Through simulations, we evaluate the efficiency of parallel algorithms under various cache configurations. Our goal is to understand the implications of these configurations and identify optimal strategies for cache utilization. The results of this study provide valuable insights for computational performance optimization in the modern era of technology.*

1. Introduction

With the knowledge developed throughout the 20th century, various technologies were created to improve application runtime. One of them is cache memory, which leverages the repetitive nature of many algorithms to optimize execution by keeping previously accessed data in low-latency memory [Handy 1998]. Despite significant advances in cache usage optimization and the implementation of parallel algorithms, there are still gaps to be filled. The complexity and diversity of cache configurations, along with the variety of algorithms and their respective resource demands, make optimization a continuous challenge [Yavits et al. 2014].

Moreover, parallel algorithms behave differently on different architectures. This behavior creates opportunities for analyses aimed at developing new algorithms and their combinations with cache hierarchies [Rattanatranurak and Kittitornkun 2020, Fang et al. 2017]. Therefore, the following question arises: How do different cache configurations impact the performance of parallel algorithms, and what are the best strategies to optimize cache usage in various scenarios?

In this context, this work proposes to investigate how cache configurations interfere with the overall performance of an application. It presents scenarios that vary in coherence protocols, cache size, and the number of cores to comparatively analyze the resulting metrics, correlating them with the nature of the executed parallel algorithms. We aim to evaluate the impact of different cache configurations on the performance of parallel algorithms through simulations. The objective is to understand how distinct configurations can influence application efficiency and identify the most effective strategies for optimizing cache usage.

Using *gem5* [Lowe-Power et al. 2020], one of the most widely used computer architecture simulators, specific configurations will be presented. The gathered metrics provides an understanding on how the simulated architectures responds to different configurations using an workload developed using OpenMP.

The contributions to the state of the art are:

- Present comparative data between different cache configurations and their correlations with parallel algorithms.
- Investigate the computational capacity provided by coherence protocols and their benefits concerning non-shared data cache structures.
- Analyze the behavior of coherence protocols in the face of different numbers of cores in multicore systems.

The rest of the work is organized as follows. Section 2 presents a literature review and related work. Section 3 presents the methodology used, as well as the algorithms and configurations. Section 4 presents the results we gathered. Finally, Section 5 presents the conclusions.

2. Background

Cache memory has become a fundamental component of contemporary computing, playing a crucial role in reducing the access time to data stored in main memory whenever the processor issues a request. This operation is based on the locality principle, which refers to the tendency of the processor to access data close to previously used data or to access recently used data, characterized respectively as spatial and temporal locality [Patterson and Hennessy 2013].

Given the needs that have arisen over time, new capabilities have been added to cache memories. The hierarchical organization in multiple levels, up to the Last Level Cache (LLC), allows more data to be stored in the cache at the cost of reduced read and write times – although still faster than accessing primary memory. Another advancement was the addition of replacement policies, which control which elements saved in the cache will be replaced in case of address conflict. These policies can consist of simple algorithms (FIFO, Random), or be based on access frequency or recency [Patterson and Hennessy 2013].

Another important characteristic of available cache models is the policies for coherence protocols. In multicore systems, each core has its own cache, which leads to an inconsistency problem since one core can alter the values contained in its cache, but these values are not updated in the other caches if there are multiple copies of these values [Stallings 2010]. One of the protocols is MESI, in which the cache receives two state bits that allow the definition of four states. Figure 1 shows an state transition diagram for MESI protocol. MESI uses the following states:

- **Modified**: The line in this cache is different from the main memory and is available only in this cache.
- **Exclusive**: The line in this cache is the same as in the main memory and is not available in any other cache.
- **Shared**: The line in this cache is present in the main memory and may be present in another cache.
- **Invalid**: The line in this cache is not valid, and the data must be fetched from the main memory.

Another existing protocol is MOESI, which is an extension of MESI, adding a new state called *Owned*. This state aims to indicate that the associated block is owned by

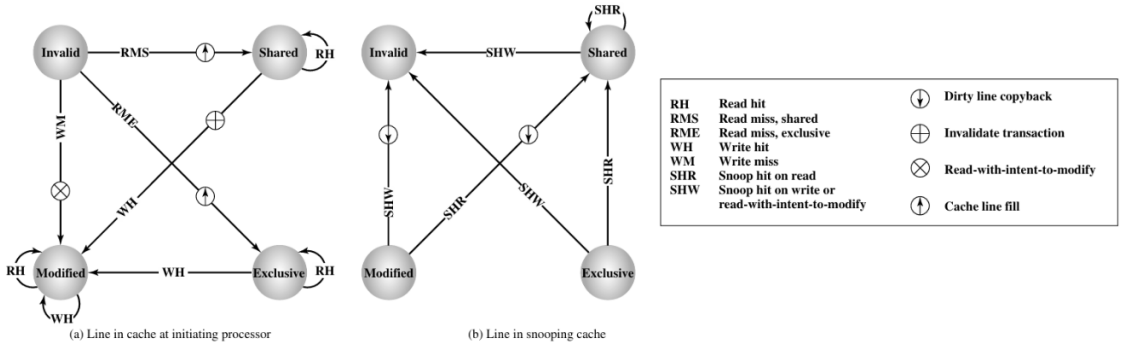

Figure 1. MESI State Transition Diagram

that cache and is outdated in the memory. When a cache has a block in the *Modified* state, in MOESI, it can be directly changed to the *Owned* state without writing back to the main memory. Other caches maintain the *Shared* state. *Owned* indicates that the original cache is the owner, and when a block is not in the cache, the owner provides it. If replaced in the cache, it is written back to the memory [Patterson and Hennessy 1990].

2.1. Related Work

Previous studies have evaluated various aspects of cache to identify optimizations in cache usage. For instance, there is an analysis of the applicability of coherence protocols in emerging architectures, such as three-dimensional DRAMs and next-generation LLCs (NG-LLC) [Zhu et al. 2021]. Based on the results, MOESI-based protocols show high latency for new LLCs, outperforming other protocols, including the one introduced by the authors, which performs 10% better than the others.

Furthermore, an evaluation of the impact of insertion policies (IP), replacement policies (RP), and prefetchers (PF) was conducted [Backes and Jiménez 2019]. In the case of IPs, there is a simplification of coherence protocols. With all data copies present in the highest-level cache, coordination to ensure data consistency is centralized, reducing the need for communication between caches of different levels. The authors evaluated various policies and concluded that there is no superior configuration; each has its benefits and drawbacks, and the other characteristics of the cache configuration should be considered to determine the best policy for the presented context.

A group of extensions of the MSI and MESI protocols were implemented, called Predictable Coherence Protocols [Kaushik et al. 2021]. The analysis of PMESI and PMSI have average slowdowns of $1.46\times$ and $1.45\times$, respectively, compared to conventional protocols. However, they performed $4\times$ better than cache bypass mechanisms that prohibit the caching of shared data in the private caches of the cores.

Last, the impact of cache associativity in performance in multicore systems was evaluated [Ramtake et al. 2020]. The scenario simulation was conducted using the Multi2Sim simulator in a quad-core configuration. The merge-sort algorithm was the workload. The best result obtained was with a 16-way associative cache, which maximized cache hits, improving overall performance.

3. Methodology

This section presents the proposed simulations for collecting performance results.

3.1. Simulated Architecture

The simulations proposed in this work are based on the x86 architecture, which uses a CISC instruction set. The x86 architecture allows for the analysis of configurations present in the market, as Intel processors use this architecture.

Eight different cache architecture organizations were simulated. They vary in the number of cores, cache size, and coherence protocols present in the cache. All of them are structured in two-level hierarchies using LRU replacement policy. All cores share the L2 cache and have a dedicated private L1 cache, which is divided into 64kB for data and 64kB for instructions. The configurations are detailed in Table 1.

3.2. Simulator

The simulator used was gem5, which operates using discrete events, meaning it has the capability to simulate the passage of time in systems, enabling the generation of data at the cycle and system level, such as TLB misses.

The simulation model used was the full system, with a Linux kernel version 4.4.186. The processor uses the out-of-order execution model with a 5-stage pipeline, which, combined with the full system, provides high temporal accuracy during the simulation.

The two coherence protocols used are based on the previous implementation by gem5. Both the MESI and MOESI protocols were modified so that the cache sizes could be configured as desired and the latencies were equalized in both protocols. The results were extracted from the execution statistics generated by gem5.

3.3. Workload Used

The workload used aim to focus more directly on the computational resources that interact with cache usage. The Pi estimation using Monte Carlo Algorithm has a low data reuse and the evaluated values are random and non-continuous. It is based on generate random points in a square with side length of *2r* centered at (0,0) with a circle of radius *r* inside it, count the points that fell inside the circle and then apply the following formulas:

When *I* represents the value of points inside the circle, and *T* represents the total number of generated points, we have:

$$q = \frac{I}{T}$$

$$\pi = 4 \times q$$

3.4. Configurations Used

The simulation setup is presented in Table 1. The configurations are identified by an ID, where the first part represents the protocol and the following numbers correspond to the core quantity, the L1 Cache Size and the L2 Cache Size, respectively.

4. Results

This section presents the results obtained from the simulations of the proposed architectures for the workload. Figure 2 shows the comparison of IPCs between each

ID	Protocol	Core Quantity	L1 Cache Size	L2 Cache Size
mesi-4-64-512	MESI	4	64 KB	512 KB
mesi-8-64-512	MESI	8	64 KB	512 KB
moesi-4-64-512	MOESI	4	64 KB	512 KB
moesi-8-64-512	MOESI	8	64 KB	512 KB
mesi-4-128-1024	MESI	4	128 KB	1024 KB
mesi-8-128-1024	MESI	8	128 KB	1024 KB
moesi-4-128-1024	MOESI	4	128 KB	1024 KB
moesi-8-128-1024	MOESI	8	128 KB	1024 KB

Table 1: Simulation setup

architecture, where IPC (Instructions Per Cycle) is a metric that indicates how many instructions a processor can execute in a single clock cycle. It can be observed that, overall, architectures with 8 cores performed better than those with 4 cores. However, the architectures *mesi-4-128-1024* and *moesi-4-128-1024* performed only 10% worse compared to *mesi-8-128-1024*, highlighting the fact that simply increasing the size of the L1 and L2 caches can compensate for the disparity in computational power. This result comes from reduced main memory communication, saving cycles.

Regarding execution time, it is observed that when the cache has a larger size, the other characteristics become less impactful on the final result. This occurs because, even with an IPC close to the average, the delay caused by primary memory access directly impacts execution time. However, with smaller caches, MOESI performs better than MESI in configurations that differ only in protocol as shown in Figure 3. This can be observed when comparing configurations such as *mesi-4-64-512* with *moesi-4-64-512* and *mesi-8-64-512* with *moesi-8-64-512*.

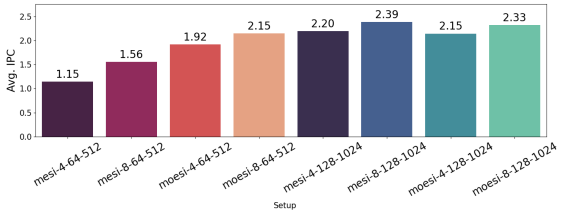

Figure 2. IPC Comparison

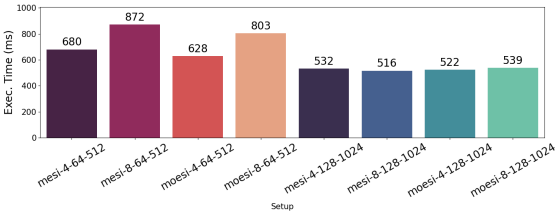

Figure 3. Exec. Time Comparison

Furthermore, mesi-4-128-1024 performed 65% better than *mesi-4-64-512*, while all other architectures performed at least 86% better when compared to the same configuration. The lower speedup is also correlated with the cache size and reduced number of cores. However, even with the least powerful configuration, it performed considerably better due to the MOESI protocol.

Figure 4 shows the results related to the L1 cache. L1 cache is divided into two types: data cache and instruction cache. It is observed that, in general, the most important factor is the cache size. The larger the cache capacity, the fewer cache misses occur, leading to improved overall performance.

The reduction in cache misses is almost linear for instructions and non-linear for data because, as the cache size increases, it can store more frequently accessed data and instructions, reducing the need to fetch from slower main memory. This can be observed when configurations *moesi-8-64-512* and *moesi-8-128-1024* are compared. Both have the

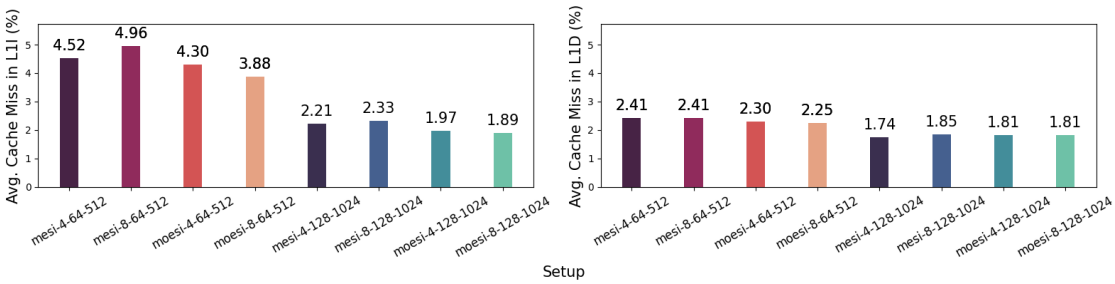

Figure 4. L1 Cache Miss Comparison

same configuration and protocol, but *moesi-8-128-1024* has double the cache size and performs 52% better when analizing the cache miss in L1I.

For the same configurations, the performance of *moesi-8-128-1024* is 20% better then *moesi-4-128-1024* when the L1D cache miss is analyzed. This behavior is the same for all configurations when the comparison is based in cache size.

Regardless of cache size and the number of cores, the MOESI (Modified, Owner, Exclusive, Shared, Invalid) protocol tends to perform better compared to the MESI (Modified, Exclusive, Shared, Invalid) protocol. *moesi-8-64-512* performs 21% better than *moesi-8-64-512* when the L1I cache miss is analyzed and 8% better when the L1D cache miss is analyzed. The behavior remains consistent when comparing moesi-8-128-1024 and moesi-8-64-512, where the performance gain in the L1I is 19% and 4% in the L1D. The larger difference in the L1I is due to the workload used, which is based on repeating the same calculations but on different data, leading to a greater reuse of instructions than data.

For larger cache sizes, the reduction in cache misses becomes even more significant. This indicates a more efficient use of the available storage capacity, as larger caches can better exploit temporal and spatial locality.

The superior performance of MOESI compared to MESI is maintained when analyzing the L2 cache, which results are presented in Figure 5. However, as the L2 cache size increases, the advantage of MOESI over MESI diminishes. This can be observed when comparing configurations *mesi-4-64-512* and *moesi-4-64-512*, where the gain is 8%, while the gain between *mesi-4-128-1024* and *moesi-4-128-1024* is only 1%. This pattern holds for other cases where the comparison is made between configurations that differ only in protocol, such as *mesi-8-64-512* with *mesi-8-64-512* and *mesi-8-128-1024* with*moesi-8-128-1024*.

This occurs because, with a larger L2 cache, the probability of cache conflicts decreases, and the additional benefits of the ‘Owned’ state in MOESI become less significant. In a shared L2 cache, which is accessed by all cores, a larger cache size means it can hold a more comprehensive set of data and instructions from multiple cores. This reduces the likelihood of cache misses and evictions, thereby minimizing the need for the advanced features of the MOESI protocol to manage coherence.

In smaller L2 caches, the ‘Owner’ state in the MOESI protocol is beneficial as it allows modified data to be transferred directly between caches without involving the main memory. This state reduces latency and coherence traffic, which is crucial when cache

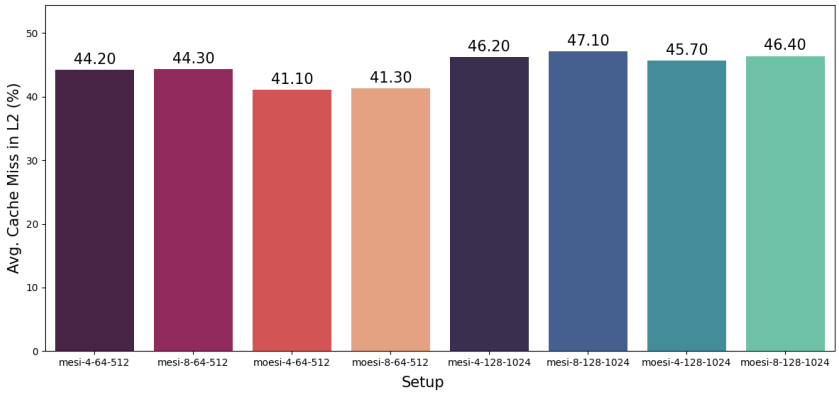

Figure 5. L2 Cache Miss

resources are limited and more frequent conflicts occur. However, as the L2 cache size increases, these conflicts become less frequent, and the necessity for direct cache-to-cache transfers diminishes.

5. Conclusion

This paper presented an analysis of cache coherence protocols in multicore systems. It demonstrated that both cache size and the choice of coherence protocol significantly impact system performance. For L1 caches, larger sizes lead to fewer cache misses and improved performance, with the MOESI protocol generally outperforming the MESI protocol due to its reduced coherence traffic and faster cache-to-cache transfers. The advantages of the MOESI protocol are especially evident in scenarios with smaller caches and higher conflict rates.

However, as we transition to analyzing the shared L2 cache, the performance gap between MOESI and MESI diminishes with increasing cache size. This is because larger L2 caches reduce the probability of cache conflicts, making the additional benefits of the 'Owner' state in MOESI less significant. The shared nature of the L2 cache means it can more effectively serve multiple cores as its size increases, reducing the need for advanced coherence mechanisms to manage data consistency.

In conclusion, the performance of cache coherence protocols in multicore systems is highly dependent on cache size and architecture. While MOESI offers advantages in smaller, private caches by reducing latency and coherence traffic, these benefits become less pronounced in larger, shared caches. This underscores the need for a balanced approach in multicore CPU design, considering both the cache architecture and coherence protocol to achieve optimal performance and power efficiency. As the number of cores in multicore CPUs continues to grow, the importance of efficient cache architectures and coherence protocols will only increase, making this an essential area of study for future processor designs.

In future research, it is important to explore how coherence protocols handle different workloads with varying characteristics, such as memory access patterns, storage requirements, and other specific needs. Additionally, examining the energy efficiency of these protocols on architectures designed for mobile devices is vital given the

rising computational demands and the need for low energy consumption. Additionally, This comprehensive evaluation will provide valuable insights into optimizing coherence protocols for diverse and demanding applications.

References

Backes, L. and Jiménez, D. A. (2019). The impact of cache inclusion policies on cache management techniques. In *Proceedings of the Int. Symp. on Memory Systems*, MEMSYS '19, page 428–438, New York, NY, USA. ACM.

Fang, J. et al. (2017). Performance optimization by dynamically altering cache replacement algorithm in cpu-gpu heterogeneous multi-core architecture. In *IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing*, pages 723–726.

Handy, J. (1998). *The cache memory book (2nd ed.): the authoritative reference on cache design*. Academic Press, Inc., USA.

Kaushik, A. M., Hassan, M., and Patel, H. (2021). Designing predictable cache coherence protocols for multi-core real-time systems. *IEEE Transactions on Computers*, 70(12):2098–2111.

Lowe-Power, J. et al. (2020). The gem5 simulator: Version 20.0+.

Patterson, D. A. and Hennessy, J. L. (1990). *Computer architecture: a quantitative approach*. MK Publishers Inc., San Francisco, CA, USA.

Patterson, D. A. and Hennessy, J. L. (2013). *Computer Organization and Design: The Hardware/Software Interface*. MK Publishers Inc., San Francisco, CA, USA.

Ramtake, D., Singh, N., Kumar, S., and Patle, V. K. (2020). Cache associativity analysis of multicore systems. In *Int. Conf. on Comp. Science, Eng. & Applications*, pages 1–4.

Rattanatranurak, A. and Kittitornkun, S. (2020). A parallel triple-pivot sorting (ptpsort) algorithm: Preliminary results. In *17th Int. Conf. on Electrical Eng./Electronics, Computer, Telecom. and I.T.*, pages 59–62.

Stallings, W. (2010). *Computer Organization and Architecture: Designing for Performance*. Prentice Hall.

Yavits, L., Morad, A., and Ginosar, R. (2014). Cache hierarchy optimization. *IEEE Computer Architecture Letters*, 13(2):69–72.

Zhu, M., Shahab, A., Katsarakis, A., and Grot, B. (2021). Invalidate or update? revisiting coherence for tomorrow's cache hierarchies. In *30th International Conference on Parallel Architectures and Compilation Techniques*, pages 226–241.