Sharif University of Technology
Department of Computer Engineering

Project Bachelor's
Computer Engineering

Reuse Distance Calculation

By

Amirreza Inanloo, Ali Majidi, Artin Barghi, Bardia Rezaei Kalantari

Supervisor

Hossein Asadi

July 2024

Abstract

In this paper, methods for calculating reuse distance in cache memory systems are explored. Reuse distance is a metric that measures the number of distinct memory references that occur between two accesses to a specific memory address. This metric is crucial for analyzing cache behavior and optimizing its performance.

Two primary methods for calculating reuse distance are introduced: the stack-based method and the tree-based method. The stack-based method, known for its simplicity but inefficiency, utilizes a stack to store observed addresses and performs stack searches and push/pop operations each time a new address is encountered. Despite its simplicity, this method has a time complexity of $O(n^2)$ and is not suitable for large inputs.

In contrast, the tree-based method calculates reuse distance more efficiently using an optimized tree structure. In this method, each tree node contains the memory address and the number of its child nodes. By employing specific techniques, the reuse distance is computed with a time complexity of $O(n \log n)$. This method, while maintaining the address order in the tree similar to the stack, offers increased efficiency and higher accuracy.

The results of this study show that employing advanced data structures can significantly enhance the calculation of reuse distance and, consequently, optimize cache memory performance. The paper provides a comprehensive comparison between the two methods, discusses their advantages and disadvantages, and offers suggestions for future research in cache system optimization.

Contents

Chapter 1

Introduction

The first chapter of this project introduces the problem, states the importance of the topic, reviews the literature, outlines the research objectives, and describes the structure of this paper.

1.1 Problem Definition

Reuse distance in cache refers to the time period between repeated accesses to a specific data item in the cache. This concept plays a critical role in improving the performance of computer systems, as cache, with its faster access compared to main memory, enhances the execution speed of programs. A shorter reuse distance indicates higher cache utilization and reduced data access time.

1.2 Importance of the Topic

The importance of examining reuse distance in cache lies in the fact that optimizing this distance can significantly enhance the performance of computer systems. Given the increasing volume of data and the complexity of programs, efficient cache utilization can greatly impact reducing delays and increasing processing speed. This is particularly important not only for ordinary users but also for data centers and large-scale systems

requiring fast processing.

1.3 Literature Review

Numerous studies have been conducted on cache optimization and data access patterns. Research has shown that by analyzing access patterns and predicting program behavior, reuse distance can be reduced. Various cache replacement algorithms have also been introduced to optimize this distance. Some studies have focused on improving predictions, while others have concentrated on optimizing cache structures.

1.4 Research Objectives

The primary objective of this research is to investigate and analyze reuse distance in cache with a focus on optimizing access patterns and reducing this distance. Specific objectives of this research include:

- Analyzing algorithms for calculating reuse distance in cache in terms of time and memory
- Developing and evaluating optimized algorithms for calculating reuse distance in cache
- Analyzing program locality based on reuse distance results

1.5 Research Structure

This thesis is presented in five chapters as follows: Chapter two discusses the basic concepts. Chapter three reviews prior work in the field of reuse distance calculation. Chapter four presents the new results obtained in this research. Chapter five provides a summary and analysis of the work done in this research.

Chapter 2

Basic Concepts

In this chapter, we will explain and introduce the concepts used in this project, including Spatial Locality, Temporal Locality, Reuse Distance, Cache, and Cache Policy.

2.1 Locality

2.1.1 Temporal Locality

Temporal Locality is one of the important principles in computer memory design, which states that if a memory location is accessed at a specific time, there is a high probability that the same memory location will be accessed again in the near future. In other words, data or instructions that have been recently accessed are more likely to be used again soon. This principle is commonly used in cache memories and memory management mechanisms to improve system performance.

2.1.2 Spatial Locality

Spatial Locality is another important principle in computer memory design, which indicates that if a memory location is accessed at a specific time, there is a high probability that nearby memory locations will also be accessed in the near future. In other words, data or instructions that belong to consecutive and close memory locations are usually

accessed sequentially. This principle is also used to improve system performance and efficiency of cache memories.

2.2 Reuse Distance

Reuse Distance refers to the number of memory accesses to unique addresses that occur between two consecutive accesses to a specific memory location. In other words, reuse distance is the number of unique memory operations performed between two consecutive accesses to a specific memory location.

2.2.1 Applications

- Cache Optimization: Analyzing reuse distance allows system designers to better understand memory access patterns and tune caches for improved performance.
- Memory Performance Prediction: By using reuse distance, the performance of programs in systems with various cache sizes can be predicted and programs can be optimized.
- Memory Allocation and Management: Reuse distance can help in optimizing memory allocation and resource management to ensure that frequently used data resides in higher levels of memory.

2.3 Cache

Cache Memory is a type of high-speed, temporary memory used in computer systems to improve access speed to data and instructions. Cache acts as an intermediary between main memory (RAM) and the processor (CPU), storing frequently or recently used data to minimize access time when needed again.

2.3.1 Cache Levels

Cache is typically organized into several levels (L1, L2, L3):

• Level 1 (L1): The closest level to the processor and the fastest cache, but with a smaller capacity.

• Level 2 (L2): Slower than L1 but with a larger capacity.

• Level 3 (L3): Slower than L2 but with a much larger capacity and shared among multiple processor cores.

Cache, as one of the critical components in modern computer system design, plays a significant role in improving system performance and overall efficiency.

2.3.2 Applications

• Increased Processor Performance: By storing frequently accessed data and instructions, the cache reduces the processor's access time to this information, thus speeding up program execution.

• Reduced Memory Access Latency: Main memory (RAM) has higher access time compared to cache. By temporarily storing frequently used data in cache, the need for frequent access to main memory is reduced, thus decreasing access latency.

• Optimized Memory Bandwidth: By reducing the number of direct accesses to main memory, cache helps optimize memory bandwidth and improves overall system performance.

• Increased Efficiency in Multiprocessor Systems: In multiprocessor systems, cache allows each processor to quickly access its required data, thus reducing contention for access to main memory.

2.4 Cache Policies

Cache Policies are a set of rules and algorithms that determine how data is stored and replaced in the cache. These policies are used to optimize cache performance and improve

overall system performance.

2.4.1 Types of Cache Policies

Replacement Policy

Least Recently Used (LRU): In this policy, data that has not been used recently is removed from the cache when new space is needed. This policy assumes that data which has not been recently used is less likely to be used again.
First In, First Out (FIFO): In this policy, data that was entered into the cache earlier is removed first when new space is needed. This policy considers the order of data entry into the cache.
Least Frequently Used (LFU): In this policy, data that has been used less frequently is removed from the cache. This policy operates based on the frequency of data usage.

2.4.2 Applications of Cache Policies

• System Performance Improvement: By using appropriate policies, cache performance can be optimized, thereby increasing data access speed and overall system performance.
• Enhanced Cache Efficiency: Effective replacement policies can help keep frequently used data in the cache, reducing the need for access to main memory.
• Energy Consumption Optimization: By reducing the number of unnecessary accesses to main memory, system energy consumption is also reduced.

Chapter 3

Related Work

3.1 Theory

Two different implementation methods are discussed in the following paper. The first is a stack-based implementation:

https://sites.cc.gatech.edu/classes/AY2013/cs7260_fall/lecture30.pdf

Algorithm:

Processing each memory address:

Search for the address in the stack: Every time you encounter a memory address, look for it in the stack.

If the address is not found in the stack:

Consider the reuse distance as infinite (∞) since this is the first time this address has been observed.

Place the new address at the top of the stack.

If the address is found: If the address is found in the stack:

Temporarily pop all elements above the target address from the stack.

Remove the target address from the stack and discard it.

Push the temporarily popped elements back onto the stack in reverse order.

Calculate the reuse distance as the number of temporarily popped elements.

Place the new address at the top of the stack.

Output: Compute the reuse distance for each memory address in the input stream. These values can be used to generate a histogram of reuse distances.

The time complexity of this program is $O(n^2)$, and it is not suitable for large inputs.

The second method is a tree-based implementation, which matches the time complexity requirement of the project, and we have implemented this second method, which will be discussed in the Results chapter.

3.2 Implementation

The following link contains C++ code where a program is written using the concept of LinkedList to compute the reuse distance:

https://drive.google.com/drive/folders/1wK6Jh80SLH5whCfKBOeW9QI7_290TkxC

Main components of the program:

The program is divided into several main parts:

Main function: Includes code for reading inputs, processing data, and writing outputs.

Linked list: For storing memory requests and managing accesses.

Reuse distance calculation method:

Reading and processing data: Data is read from input files. The input files contain memory addresses and the number of reuse occurrences.

Storing requests in the linked list: Each memory request is added to the linked list.

Removing and calculating reuse distance: If a memory address is present in the linked list, it is removed, and the time distance is calculated. Also, statistics related to the number and size of requests are updated.

Storing statistics in a vector: The calculated statistical values are stored in a vector.

Time and space complexity analysis:

Time Complexity:

Reading data from files: $O(n)$ where n is the number of lines in the input file.

Searching in the linked list: $O(n)$ in the worst case for each search.

Adding and removing in the linked list: $O(1)$ for adding and $O(n)$ for removing.

Calculating statistics: $O(nlogn)$ for sorting and $O(n)$ for computing mode and median.

Thus, the overall time complexity is $O(n^2)$.

Space Complexity:

Storing data in vectors and linked list: $O(n)$

Storing statistics in a vector: $O(n)$

Therefore, the space complexity is $O(n)$.

In summary, this program is designed to compute the reuse distance of memory and uses a linked list to manage memory requests. Its time complexity is $O(n^2)$ and its space complexity is $O(n)$. The program calculates memory access statistics using various functions and stores them in an output file.

Chapter 4

New Results

4.1 Algorithm

According to the paper referenced in the previous section, a method for calculating reuse distance using a specific type of binary tree was introduced, which has a time complexity of $O(n \log n)$ and a space complexity of $O(n)$.

In this algorithm, each node holds 6 values:

1. Address value
2. Height of the subtree
3. Number of nodes in the subtree
4. Pointer to the left child if it exists
5. Pointer to the right child if it exists
6. Pointer to the parent

We also need two hash maps:

1. From address to the corresponding node in the tree
2. From address to a list of reuse distances

Algorithm:

1: Extract all addresses from the file and sort them by access time (initially sorted, so this step is not needed here).

2: For each address:

2-1: Check if the address is mapped to a node in the hash map. If not, add a new leaf to the left of the leftmost leaf in the tree and update the subtree node counts, while also balancing the tree with rotations. In the reuse distance hash map, map the address to an empty list.

2-2: If a node exists:

2-2-1: First, find the position of that node in the inorder traversal (starting from zero) and add it to the list corresponding to that address. For this, use the number of nodes in the subtree stored for each node.

2-2-2: Delete that node and add a new leaf to the left of the leftmost leaf in the tree. In both steps, update the subtree node counts and balance the tree with rotations. Finally, update the hash map with the node corresponding to that address.

2-3: Move to the next address and go to step 1-2.

3: Finally, for each address, we have a list of reuse distances for that address, from which we can compute any statistical data.

The source code is available at this link.

4.2 Time and Space Complexity

The tree used is balanced, so its height is of order log(n). Since all operations implemented for the tree are linear with respect to the height of the tree, the operations are of order log(n).

In this algorithm, for each address, a fixed number of operations are performed on the tree, and considering n addresses, the time complexity of the algorithm is $O(n \log n)$.

For memory, there are 2 hash maps with order n, so the space complexity of the algorithm is $O(n)$.

4.3 Output of the Code for Various Trace Files

The results obtained from running the code on Alibaba trace files are available at this link.

4.3.1 Output A669.csv

```
access count: 27244833
distinct accesses: 179031
average access count per offset: 152.17941585535465
median access count per offset: 6
std access count per offset: 809.72370309121588320453554455642352037874925767859
max access count per offset: 11421
count of offsets with no reuse: 15908
average reuse distances: 7292.39585307688277627982352047059237335734592309 51
median reuse distances: 3153.0
std reuse distances: 19779.269581852503085076357696661465178146827586377
min reuse distances: 0
max reuse distances: 178209
time: 1166.8812279701233
```

4.3.2 Output A129.csv

```
access count: 17712493
distinct accesses: 2999197
average access count per offset: 5.905745104439622
median access count per offset: 1
std access count per offset: 261.160546083141904134044895966055148431038931594 17
max access count per offset: 250614
count of offsets with no reuse: 1623383
average reuse distances: 235256.84283759396942738051351648196298096633140528
median reuse distances: 25003.0
std reuse distances: 460156.985737916017248777798260386003552922049390 04
min reuse distances: 0
max reuse distances: 2973720
time: 842.1423711776733
```

4.3.3 Output A108.csv

```
access count: 20205318
distinct accesses: 1658530
average access count per offset: 12.182666578234944
median access count per offset: 5.0
std access count per offset: 170.99595272635280878196144557379217091445864979536
max access count per offset: 111784
count of offsets with no reuse: 372946
average reuse distances: 324007.16504733865508140816620106942506702508272591
median reuse distances: 409722.0
std reuse distances: 240484.38786791484168973727162887005639404949224929
min reuse distances: 0
max reuse distances: 1656075
time: 1173.313912153244
```

4.3.4 Output A42.csv

```
access count: 5087932
distinct accesses: 154348
average access count per offset: 32.964029336304975
median access count per offset: 26.0
std access count per offset: 416.71339384554940249076810658630344462494839560164
max access count per offset: 89662
count of offsets with no reuse: 7768
average reuse distances: 65486.807471404155680738384103726621458152937094007
median reuse distances: 73544.0
std reuse distances: 39501.64869630910653931444186379137526391956198209 4
min reuse distances: 0
max reuse distances: 154114
time: 275.7883653640747
```

Chapter 5

Conclusion

5.1 Temporal Locality

Temporal locality refers to the access pattern of data over time and the time interval between repeated accesses to a specific data item. Based on the provided metrics, a more detailed analysis of temporal locality for each file can be provided.

5.1.1 File `A42.csv`:

- Mean reuse distance: `65486.8`
- Median reuse distance: `73544`
- Standard deviation of reuse distances: `39501.6`
- Minimum reuse distance: `0`
- Maximum reuse distance: `154114`

The `A42` file has a relatively high mean and median reuse distance, indicating that accesses to a specific data item occur at longer time intervals. This could be due to accessing different data at various time intervals. The high standard deviation also indicates significant variation in reuse distances.

5.1.2 File `A108.csv`:

- Mean reuse distance: `324007.2`
- Median reuse distance: `409722`
- Standard deviation of reuse distances: `240484.4`
- Minimum reuse distance: `0`
- Maximum reuse distance: `1656075`

The `A108` file has a very high mean and median reuse distance, indicating weaker temporal locality. Accesses to specific data items are rarely repeated, and the time intervals between repeated accesses are very large. The high standard deviation also indicates a very high variation in reuse distances.

5.1.3 File `A129.csv`:

- Mean reuse distance: `235256.8`
- Median reuse distance: `25003`
- Standard deviation of reuse distances: `460156.9`
- Minimum reuse distance: `0`
- Maximum reuse distance: `2973720`

The `A129` file has a relatively high mean reuse distance, but the median reuse distance is lower compared to the mean, indicating that most accesses occur at shorter intervals, but some accesses have very long intervals that raise the mean. The very high standard deviation also indicates significant variation in reuse distances.

5.1.4 File `A669.csv`:

- Mean reuse distance: `7292.4`
- Median reuse distance: `3153`

- Standard deviation of reuse distances: `19779.3`
- Minimum reuse distance: `0`
- Maximum reuse distance: `178209`

The `A669` file has very low mean and median reuse distances, indicating excellent temporal locality. Accesses to specific data items occur frequently and at shorter intervals. The lower standard deviation also indicates less variation in reuse distances.

5.1.5 Conclusion

- `A669` has the best temporal locality. The low mean and median reuse distances and relatively low standard deviation indicate frequent and close proximity accesses to specific data items.
- `A42` has good temporal locality but not as good as `A669`. The higher reuse distances indicate less frequent accesses to the data.
- `A129` has weaker temporal locality compared to `A42` and `A669`. The high mean and very high standard deviation indicate significant variation in reuse distances.
- `A108` has the poorest temporal locality. The very high mean and median reuse distances and large standard deviation indicate very scattered and irregular accesses to the data.

5.2 Spatial Locality

To analyze spatial locality, specific information about memory access patterns and how data is distributed in memory is required. The provided metrics mostly pertain to temporal locality and cannot directly determine spatial locality. Below are the reasons why this data is insufficient for conclusions about spatial locality:

1. **Lack of Information About Access Order:** Spatial locality refers to accessing data that is near each other in memory. To analyze this, we need to know how data accesses occur in sequence and whether adjacent data in memory are accessed consecu-

tively. The provided data only gives information about the number of accesses and time intervals between accesses, not the exact order of accesses.

2. **Lack of Information About Physical Data Locations:** To analyze spatial locality, we need to know where data are located in physical memory. Without information about the memory addresses where data reside, it cannot be determined whether accesses are to nearby data. The provided information does not include physical data locations in memory.

3. **Focus on Time Intervals:** The provided metrics focus more on time intervals between repeated data accesses, such as the mean and median reuse distances. These metrics refer more to temporal locality and show how access to data is distributed over time.

4. **Lack of Spatial Locality Metrics:** To analyze spatial locality, metrics such as the ratio of consecutive accesses to nearby data or the mean distance between consecutive memory addresses are required. None of the provided metrics offer this type of information.

5.2.1 Conclusion

Given the above reasons, the provided data can only assist in analyzing temporal locality and cannot provide accurate conclusions about spatial locality. For a more precise analysis of spatial locality, additional information and specific metrics are needed to show the distribution and access order of data in memory.