Analysis of Hardware Impact on Software Performance

Yeshwanth Sai Kandula[1], Bindu Bhargavi Madireddy[2], Sourab B R[3]

[1]Dept. of Computer Science and Engineering, Amrita School of Engineering, Amritapuri, Kerela, India
[2]Dept. of Electronics and Communication Engineering, DVR & Dr. HS MIC College of Engineering, Kanchikacherla, Andhra Pradesh, India
[3]Dept. of Information Science and Engineering, Dayananda Sagar College of Engineering, Bengaluru, Karnataka, India

---***---

Abstract - *In order to build applications that are fast and reliable, we generally consider having efficient algorithms and programming is enough. But it is also vital to understand the underlying hardware and the various costs associated with it while designing software. If we know the fundamentals of programming and algorithms very well, it is easy to tell that quicksort is quicker than bubble sort. However, sometimes it is not enough to analyze and understand why some programs perform well or why they don't. Sometimes, we also need to understand the hardware and how it affects the efficiency of a program. This paper analyzes and examines how various memory and cache-related aspects affect the performance of a C++* program.

Key Words: **software, performance, efficiency, hardware, impact, cache, memory, data.**

1. INTRODUCTION

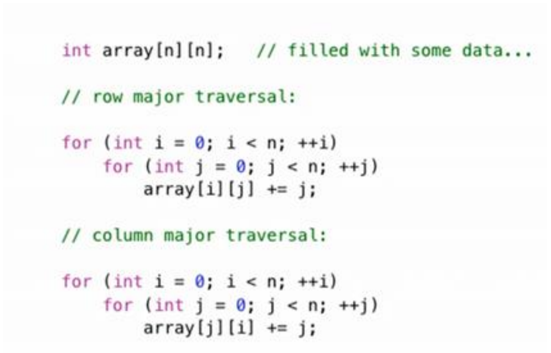
In these modern times, software applications have become an integral part of our day-to-day life. For most necessities, they have become a simple, viable, and flexible solution.

When a software program is being developed, we generally emphasize choosing relevant algorithms, design patterns, software architecture and implementing them efficiently in the desired programming language. However, we often fail to consider the design of the underlying hardware of systems for which we are developing the program. Modern computer architectures have many aspects that can impact the performance of an object-oriented program, such as cache locality, cache coherence, true and false sharing between CPU cores, memory alignment, branch prediction, instruction pipeline, and some more. Understanding these fundamentals will help determine the reasons behind the program's underwhelming metrics when it is profiled. Over the subsequent sections, we illustrate how some of these factors affect the performance of C++ programs with some examples.

2. ARRAY TRAVERSALS

2.1 Row major vs. Column major

In general, a 2D array can be conventionally traversed in two ways: 1. row-by-row (row-major) 2. column-by-column (column-major).



```
int array[n][n];   // filled with some data...

// row major traversal:

for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
        array[i][j] += j;

// column major traversal:

for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
        array[j][i] += j;
```

Fig-1: Row major and column major traversal of a 2D array

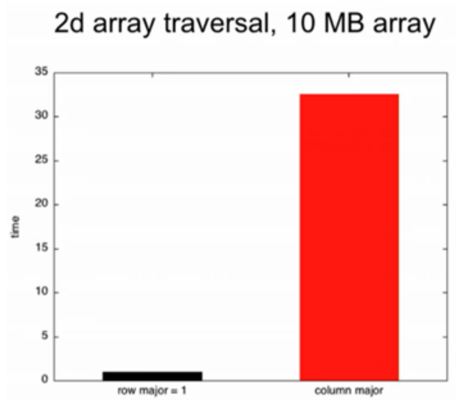


Fig-2: Metrics for row-major & column-major traversal

Upon profiling these two traversals, it is observed that time taken by column-major traversal is ~32x times of row-major traversal (Fig-2). Even though both approaches have the same number of iterations and memory access requests, we can witness a significant difference in the time taken.

Whenever the CPU performs a read operation, the memory is fetched into the cache. Fundamentally, caches read/write in the units of cache lines[1]. For a cache line size of 64 bytes, eight elements of an 8-byte integer array can fit into a single cache line (8 * 8 bytes = 64 bytes) [2]. Therefore, when the first element of the array is requested for access, a 64-byte cache line is loaded onto the cache. Due to this, the consecutive seven elements are also loaded. Hence, in row-major traversal, there are fewer cache misses as successive indexes are accessed. However, in the case of column-major traversal, the consecutive requested elements might not be available in the same cache line and thus will result in frequent cache misses. Therefore, scanning through contiguous memory is much faster compared to the latter.

2.2 Row major vs. Column major on different array sizes

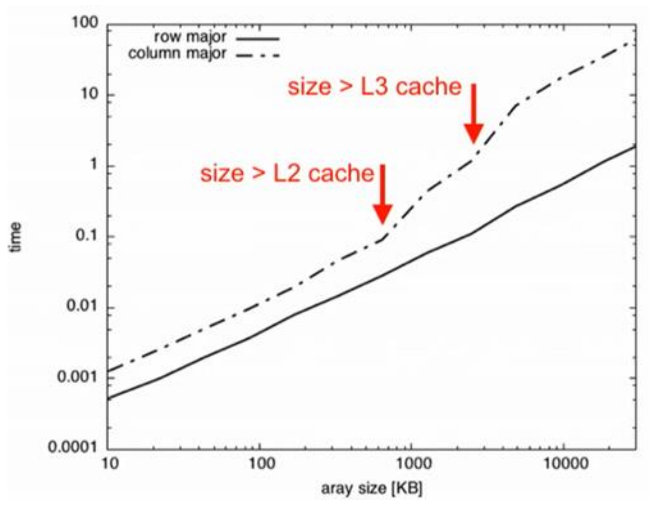
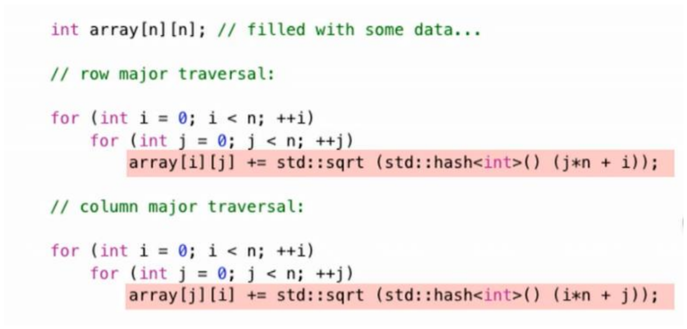


Fig-3: Metrics for row-major & column-major traversal on 2D arrays of different sizes.

When the same tests are profiled, but by varying the size of the array, the results are as shown in Fig-3. The graph shows two sudden elevations when the array size exceeds certain limits. Here, the first limit is the size of the L2 cache, and the second is of the L3 cache. In the test case, as the array grows larger, it will at some point exceed the L2 and L3 cache sizes. In section 2.1, it was concluded that column-major suffers from frequent cache misses and hinders performance. In a similar fashion, when the array size exceeds L2 cache size and can no longer fit in it, the consecutive element that the CPU wants to access during column-major might not be available even in the L2 cache. This happens frequently and thus will result in more L2 cache misses. It is known that the higher the cache capacity, the higher the latency[3]. Hence, an L2 cache miss costs more than an L1's, and an L3 cache miss costs more than an L2's. Therefore, a sharp rise occurs when the size exceeds L2 cache capacity. Similarly, when the array can no longer fit into the L3 cache size, the number of L3 cache misses increases; Hence the hike in the graph. Whereas, for row traversal, the graph's gradient remains consistent and doesn't suffer from this problem as the CPU is accessing contiguous memory blocks.

2.3 Bound by computation vs. Bound by data access



```
int array[n][n]; // filled with some data...

// row major traversal:

for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
        array[i][j] += std::sqrt (std::hash<int>() (j*n + i));

// column major traversal:

for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
        array[j][i] += std::sqrt (std::hash<int>() (i*n + j));
```

Fig-4: 2D traversal with some work

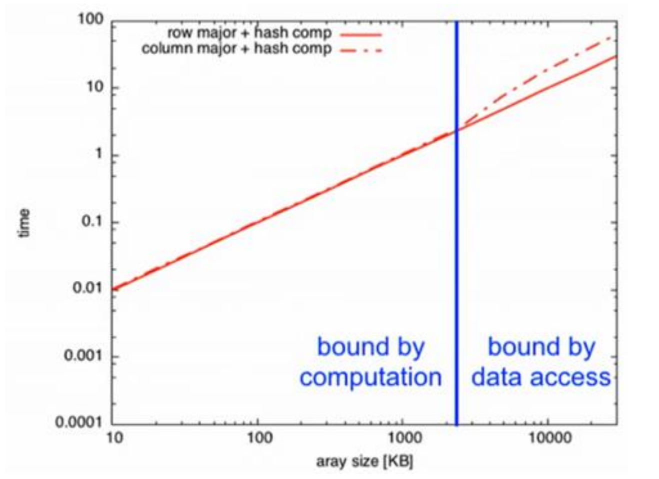


Fig-5: Bound by computation vs. bound by data access

The Fig-4 sample is a perfect example that differentiates between performance bounded by computation and performance bounded by data access.

We've observed and understood that column-major traversal is way worse when compared with row-major. However, in Fig-5, it is noticed that the graph for both cases remained almost the same for most of the array sizes until it crossed a specific limit. The reason is, the time taken by computation (i.e., the square root operation over a hashed value) is way higher that it overshadows the cost incurred due to cache misses during column-major. But, after a specific array size, the column-major traversal takes a hit in performance while row-major remains consistent.

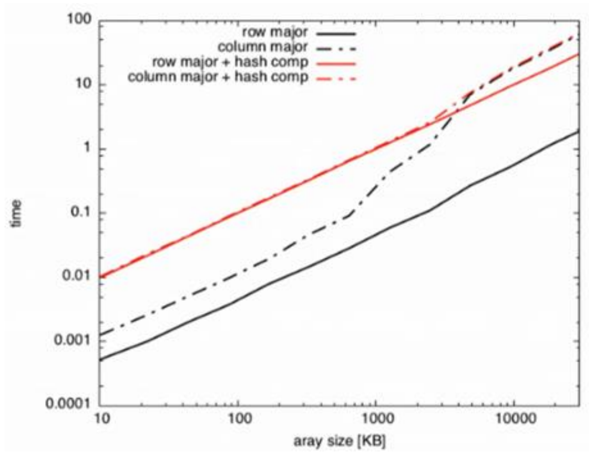
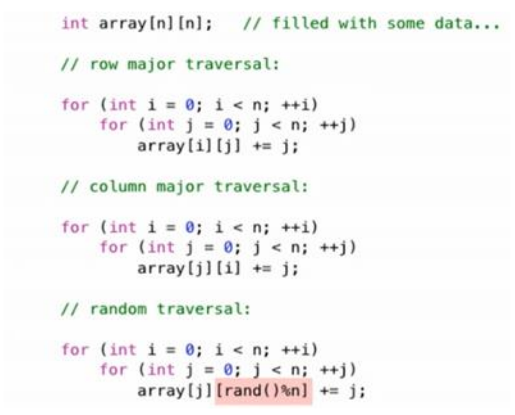


Fig-6: Illustrating scenarios of Fig-3 & Fig-5 together.

This is because post this critical point (i.e., array size > L3 cache size), the cost incurred to cache misses (L3 level) is significantly high that the memory access takes more time than the computation itself. That is when the performance is said to be bound by data access. In such cases, no matter how much we improve our algorithm of square root and hashing, the performance hindrance remains the same, as the root cause of the problem is the way the data is accessed.

2.4 Row major vs. Column major vs. Random access



```
int array[n][n];   // filled with some data...

// row major traversal:

for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
        array[i][j] += j;

// column major traversal:

for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
        array[j][i] += j;

// random traversal:

for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
        array[j][rand()%n] += j;
```

Fig-7: Sample code for row, column & random traversal.

From the metrics (Fig-8) of random index access traversal, it was observed that the random access traversal is taking much more time than column-major traversal.

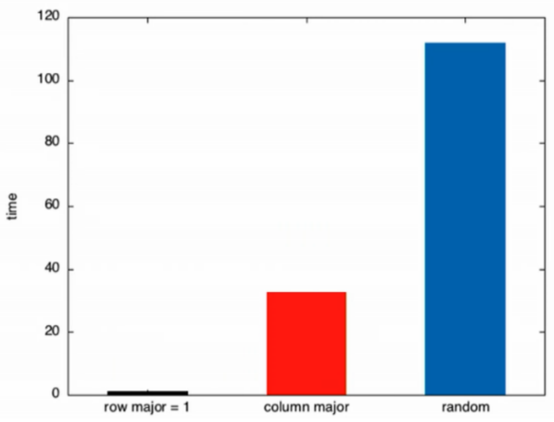


Fig-8: Metric results of code sample shown in Fig-7.

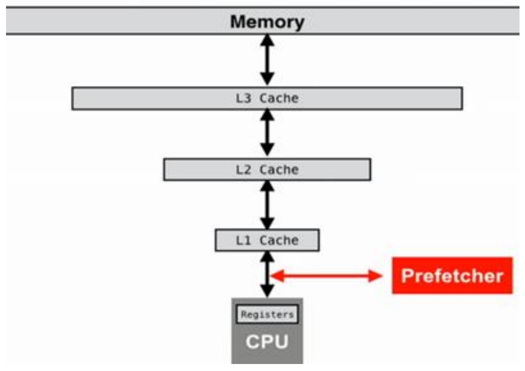


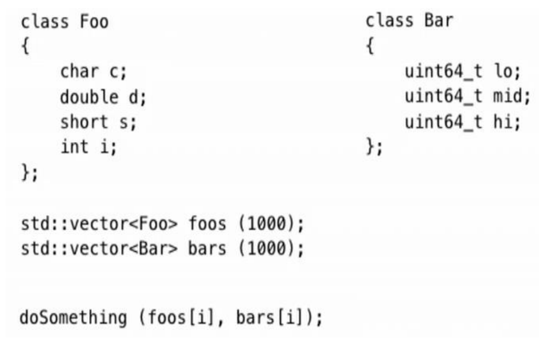
Fig-9: Visualization of Prefetcher.

This surfaced the significance of the prefetcher, which exists between the CPU and cache, continuously monitors all the traffic[4]. Whenever it notices a consistent pattern, it prefetches the data by dynamically predicting what the CPU might do next. In column-major, the jump between two consecutive data access remains the same. However, random access rules out any such pattern and inherently suffers from a significant number of cache misses. As a result, we observe that random access performs considerably slower than column-major access.

3. LOCALITY OF REFERENCE

3.1 Spatial Locality

Given the structures, Foo and Bar, as shown in Fig-10, if both these structures are frequently accessed consecutively, then it is better advised to make a structure of them together and store them as elements in data structures as shown in Fig-11. This results in Foo and Bar staying close to each other in the memory, thereby giving better spatial locality[5].



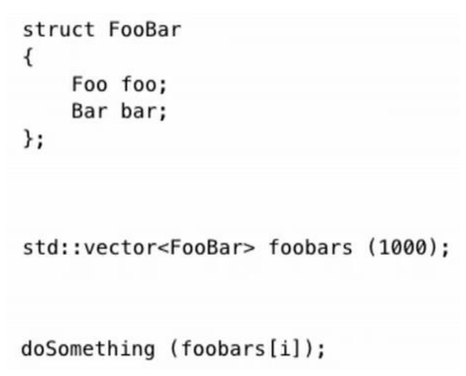
```
class Foo
{
    char c;
    double d;
    short s;
    int i;
};

class Bar
{
    uint64_t lo;
    uint64_t mid;
    uint64_t hi;
};

std::vector<Foo> foos (1000);
std::vector<Bar> bars (1000);

doSomething (foos[i], bars[i]);
```

Fig-10: Arrays of Foo and Bar structures



```
struct FooBar
{
    Foo foo;
    Bar bar;
};

std::vector<FooBar> foobars (1000);

doSomething (foobars[i]);
```

Fig-11: Array of combined structure FooBar

3.2 Instruction Order

Fig-12 and Fig-13 depict operation instructions executed by the CPU during a time interval on three variables represented by green, orange, and blue. In the figures, a green block indicates an operation executed by the CPU on the green variable, and similarly for orange and blue blocks.

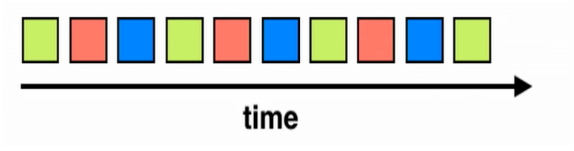


Fig-12: Inefficient instruction order

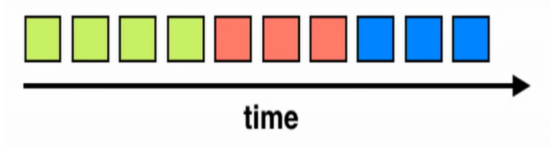


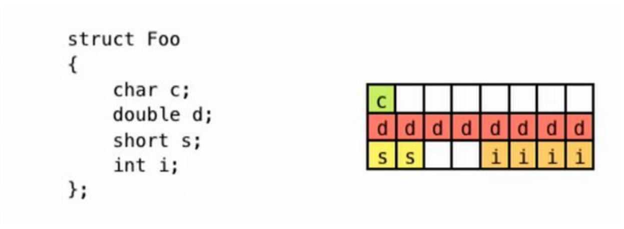
Fig-13: Efficient instruction order

In Fig-12, by the time the CPU reaches the 2nd operation on the green, the green variable may have already been replaced by another data in caches and registers[6]. This results in an increased probability of cache miss. Therefore, unless the second operation on the green is dependent on the first operation on orange, the pattern shown in Fig-13 should be followed to increase the cache hit rate.

4. MEMORY ALIGNMENT

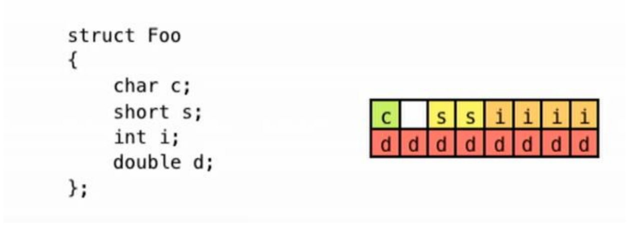
4.1 Relationship between the order of declaring data members and size of a struct

A sizeof (Struct) depends on how the data members are ordered and their respective sizes. Memory of any data type needs alignment, which depends on the system's architecture[7]. In this section, an x64 architecture is considered to illustrate a few examples.



```
struct Foo
{
    char c;
    double d;
    short s;
    int i;
};
```

Fig-14: Random ordering of data members in struct Foo



```
struct Foo
{
    char c;
    short s;
    int i;
    double d;
};
```

Fig-15: Declaring data members in the increasing order of their size

In Fig-14, although the actual memory required by "c + d + s + i" is only 15 bytes, the sizeof (Foo) results in 24 bytes. This is because variable 'c' occupies the first 8-bits of available 64-bits, and since the double 'd' needs 64 bits of aligned space, it occupies the following 64-bit aligned block, leaving a 7-byte hole in-between. Due to such data alignment, the total size of Foo becomes 24 bytes. However, in the case of Fig-15, as the data is declared in increasing order of their sizes, the total length of Foo becomes only 16 bytes. Thus, it is beneficial to structure the data members of a class/structure in increasing order of their sizes.

4.2 Data Packing

In Fig-15, a memory hole is noticed between the c and s variables. To avoid them, the 'packing' attribute can be used, as shown in Fig-16[8]. However, the way the packed data is organized and accessed varies between architectures[9, 10, 11]. For example, some architectures do the packing as indicated below in Fig-16, where the 1st byte of 'd' is placed in the last available byte of the first 8-byte line, and the rest set in the next available 7-byte aligned block.

```
struct Foo
{
    char c;
    short s;
    int i;
    double d;
}
__attribute__((packed));
```

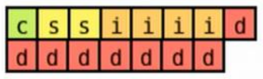


Fig-16: Memory visualization of a packed structure

In some architectures, every time an operation on 'double d' is performed, the access visualization is as shown in Fig-17. Some architectures have to read two 8-byte aligned lines, shift them, combine them, and finally operate on them. Simple memory access has now become much more complex and costly now.

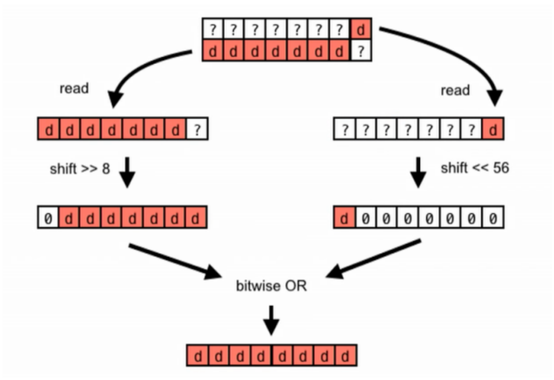


Fig-17: Visualization of unaligned memory access

When packing is used and tested on Core i7 and Core 2 Duo (Fig-18), the Core i7 system's execution is slightly faster when the data is packed. But on the Core 2 Duo, the performance takes a huge hit. As most software is generally used on diverse system technologies with varied underlying architectures, it is advised to align the data manually and not force-pack the structures. This will help avoid unforeseen costs in simple memory access operations.

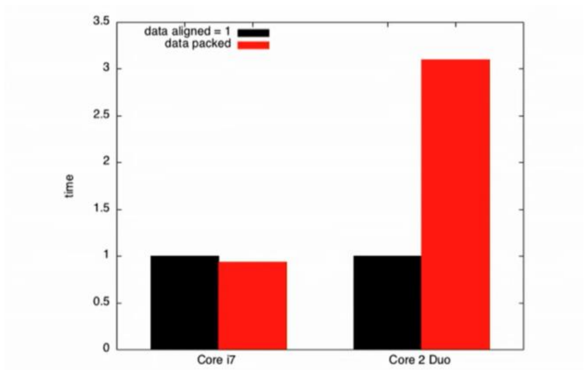


Fig-18: Aligned vs. packed data access.

5. CACHES IN MULTI-THREADING

5.1 True Sharing

```
void work (std::atomic<int>& a)
{
    for (int i = 0; i < 10000; ++i)
        a++;
}

void test()
{
    std::atomic<int> a;   a = 0;

    std::thread t1 ([&]() { work (a); });
    std::thread t2 ([&]() { work (a); });
    std::thread t3 ([&]() { work (a); });
    std::thread t4 ([&]() { work (a); });

    t1.join(); t2.join(); t3.join(); t4.join();
}
```

Fig-19: Code sample to illustrate true sharing

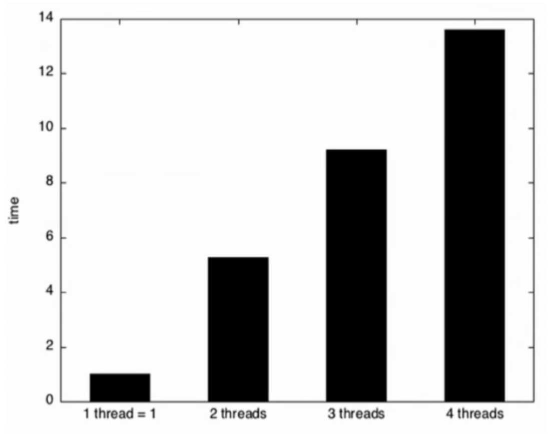


Fig-20: Metric results of code presented in Fig-19.

It is known that usage of mutexes involves kernel-level calls and context switches, which are costly[12]. But sometimes std::atomic can also be slower. In the example shown in Fig-19, 'work' is considered as the job assigned to each new thread created. If it takes Y (ms) for a single thread to complete this job, then it is expected that the time taken by 4-threads on a 4-Core machine also be nearly Y (ms), as each thread has a core each and can run parallelly. However, the results in Fig-20 convey differently.

Most of today's modern multi-core systems are designed in such a way that different cores are assigned with fully or partially separate caches. In Fig-21 example, each core has an L1 cache, while they all have shared L2 and L3 caches.

Caches fundamentally operate on the granularity of cache lines and not in terms of bytes. Hence, whenever some memory has been modified by a processor, all the cache lines of other cores' L1 caches that hold this memory block will get invalidated[13]. Therefore, the variable 'a' has to be re-fetched into the L1 cache for the other processors to continue their operations.

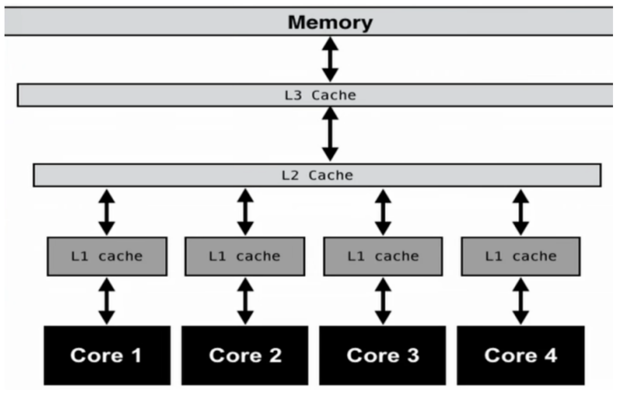


Fig-21: Each core has its own L1 Cache.

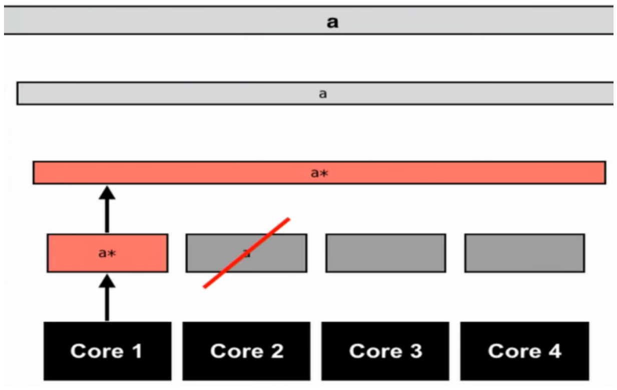


Fig-22: Visualization of cache status when core1 had modified the value of 'a' – Part-1.

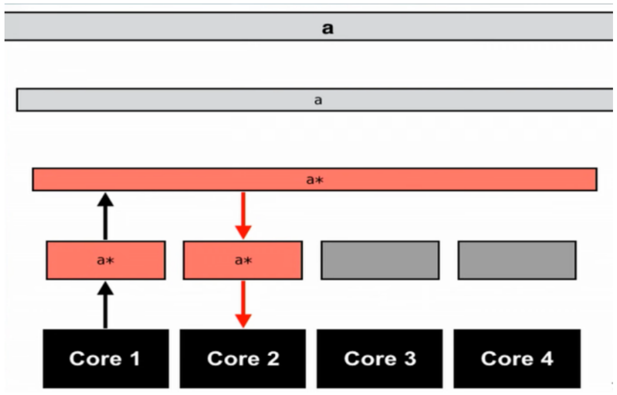
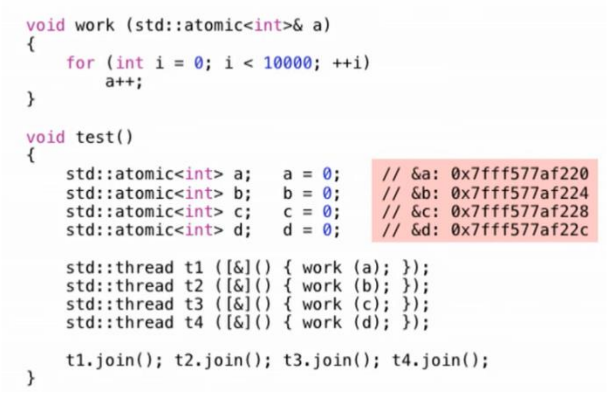


Fig-23: Visualization of cache status when core1 had modified the value of 'a' – Part-2.

In the Fig-19 example, when thread 't1' running on Core1 performed an atomic increment operation, the cache lines on the other three cores will get invalidated and have to be re-fetched for them to operate on updated memory. Such behavior is thus leading to a significant decrease in performance.

5.2 False Sharing



```
void work (std::atomic<int>& a)
{
    for (int i = 0; i < 10000; ++i)
        a++;
}

void test()
{
    std::atomic<int> a;   a = 0;   // &a: 0x7fff577af220
    std::atomic<int> b;   b = 0;   // &b: 0x7fff577af224
    std::atomic<int> c;   c = 0;   // &c: 0x7fff577af228
    std::atomic<int> d;   d = 0;   // &d: 0x7fff577af22c

    std::thread t1 ([&]() { work (a); });
    std::thread t2 ([&]() { work (b); });
    std::thread t3 ([&]() { work (c); });
    std::thread t4 ([&]() { work (d); });

    t1.join(); t2.join(); t3.join(); t4.join();
}
```

Fig-24: Code sample to demonstrate false sharing

In Fig-24, there are four integer variables, each taking up to 4 bytes of memory. Considering these variables occupied four consecutive 4-byte blocks in the memory, all four values are likely to end up on the same cache line. It is apparent that these four variables are independent of each other and that each thread's operations in 'work' have no dependency on what is happening in other threads. Yet, they falsely assume they are dependent[14, 15].

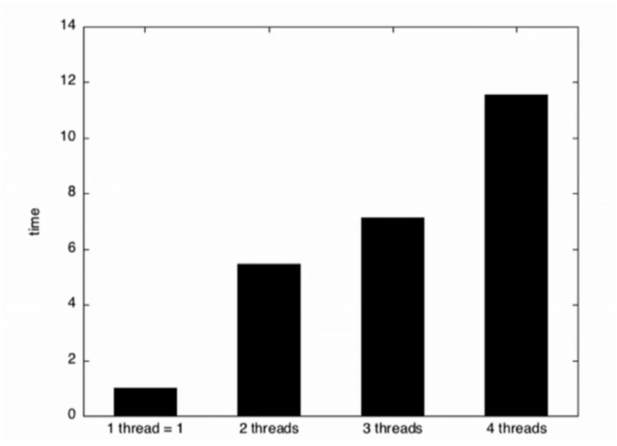
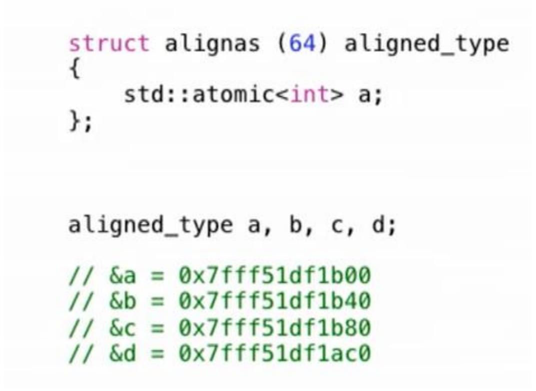


Fig-25: Metrics visualization of running 'work' function of Fig-24 with a different number of threads simultaneously.

As explained in section 5.1, Caches operate on the granularity level of cache lines. Now, every time a thread running on a core completes an atomic increment on its respective variable, this invalidates the cache line that holds this variable in all other cores. Incidentally, the other three variables also exist on the same cache line. Hence, the other cores face a cache miss, and the memory still has to be re-fetched into caches. Due to this, all four cores suffer from frequent cache misses, even though each thread is operating on independent variables. Hence, the performance is almost as bad as it was in the case of true sharing (evident by comparing results in Fig-20 and Fig-25).



```
struct alignas (64) aligned_type
{
    std::atomic<int> a;
};


aligned_type a, b, c, d;

// &a = 0x7fff51df1b00
// &b = 0x7fff51df1b40
// &c = 0x7fff51df1b80
// &d = 0x7fff51df1ac0
```

Fig-26: Visualization of memory addresses of variables when alignas is used.

However, when these variables are aligned against the cache line size (Fig-26), each of the a, b, c, d variables occupy different cache lines, thereby removing the possibility of false sharing. Hence, the time taken to complete "work" by one thread and four threads remained the same on a 4-core machine (Fig-27).

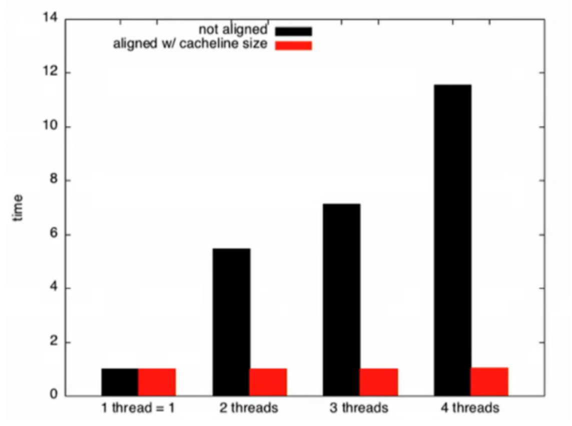


Fig-27: Not aligned vs. cacheline aligned.

6. CONCLUSION

From these observations and analysis of the gathered data, it is evident that the memory model and cache designs of hardware have a significant impact on the performance of a software program. Considering these factors and implementing cache-aware programming will significantly help develop world-class software products and services.

REFERENCES

[1] Sae-eung, Suntorn, "Analysis of False Cache Line Sharing Effects on Multicore CPUs" (2010). Master's Projects.

[2] Radu Rugina, Martin C. Rinard, "ACM Transactions on Programming Languages and Systems," Vol. 27, No. 2, March 2005, Pages 185–235.

[3] A Hartstein, V Srinivasan, Thomas Puzak, P. G. Emma, "On the Nature of Cache Miss Behavior: Is It√ 2," CF '06: Proceedings of the 3rd conference on Computing frontiersMay 2006 Pages 313–320.

[4] Fredrik Dahlgren, Michel Dubois, Per Stenstrom, F. Dahlgren, M. Dubois, and P. Strenstrom, "Fixed and adaptive sequential prefetching in shared memory multiprocessors," 1993 International Conference on Parallel Processing, pp. 56-63, August 1993.

[5] Ali Mahjur, A.H. Jahangir, Amir Gholamipour, "On the performance of trace locality of reference," Performance Evaluation Volume 60, Issues 1–4, May 2005, Pages 51-72.

[6] Alan Jay Smith, "Cache Memories," ACM Computing Surveys, Volume 14, Issue 3, Sept. 1982, pp 473–530.

[7] Nikeeta R. Patel "Data Structure Alignment," International Journal of Engineering Trends and Technology (IJETT), V45(7),338-340 March 2017. ISSN:2231-5381.

[8] Website-https://docs.oracle.com/cd/E19205-01/820-7599/giqdb/index.html

[9] Intel® 64 and IA-32 Architectures Optimization Reference Manual.

[10] Website-https://developer.arm.com/documentation/100748/0616/Writing-Optimized-Code/Packing-data-structures.

[11] Webstie-https://www.iar.com/knowledge/support/technical-notes/compiler/accessing-unaligned-data/

[12] J. Torrellas, H.S. Lam, J.L. Hennessy, "False sharing and spatial locality in multiprocessor caches," IEEE Transactions on Computers, Volume: 43, Issue: 6, Jun 1994.

[13] Edaqa, How does a mutex work? What does it cost? Website-https://mortoray.com/2019/02/20/how-does-a-mutex-work-what-does-it-cost/

[14] Manoj Kumar PA, "A Survey of Cache Coherence Protocols in Multiprocessors with Shared Memory," Proc. of the Intl. Conf. on Advances in Computer Science and Electronics Engineering, 2012. pp 384-388.

[15] William J. Bolosky, Michael L. Scott, "False Sharing and its Effect on Shared Memory," 4th Symp. on Experiences with Distributed and Multiprocessor Systems (SEDMS), San Diego, CA, Sep. 1993.