Effect of Cache Memory Access Locality on Program Performance in Microprocessors

Xiana Carrera Alonso, Ana Carsi González

Computer Architecture
Group 01
xiana.carrera@rai.usc.es | ana.carsi@rai.usc.es

April 8, 2024

Abstract

With the purpose of minimizing latency in memory accesses, multiple studies have aimed to identify the most crucial cache performance metrics. Factors such as access locality, memory hierarchy organization, and prefetching have been proven to affect efficiency. To clarify this aspect and proof its significance, this report studies the effect of the choice of the access pattern to the elements of an N-dimensional vector on the time cost of reading, as well as the effects of physical parameters of the processor and cache memory.

Keywords: Memory hierarchy, cache, temporal locality, spatial locality, prefetching, efficiency, cache miss, reduction. . .

I. Introduction

This document presents an analysis of the influence of the locality of data references on cache memory access times. Specifically, the alterations caused by different degrees of spatial and temporal locality in accessing the elements of a floating point data array in a reduction operation are studied.

In the evaluation of the results, concepts such as the organization of the memory hierarchy and the cache memory have been fundamental (for which the subject manual, [1] has been used as a reference , as well as the Computer Fundamentals bibliography book, [2]). Likewise, it has been necessary to understand the nature of cache misses ([3]), the foundations of the locality principle (with particular emphasis on spatial locality, being of special interest as a reference the analysis of [4]), the properties of prefetching operations (based on the guidance in [1]), and the effects of various code compilation options (see [5]).

Ultimately, the purpose would be to determine to what extent different program parameters, such as the separation of terms, array size, number of lines accessed, test execution environment, etc., impact the code efficiency. All of this will be examined from the perspective of the technical specifications of the testing equipment. A methodology based on the repetition of experiments for different initial conditions will be employed, filtering them by removing outliers and interpreting the final results graphically.

The structure of the document begins in section *II. Test methodology*, by a description of the static organization of the program, indicating the meaning and characteristics of its different components, and justifying the decisions that have a subsequent impact on the measurements. Next, in section *III. Execution*, the dynamic circumstances of carrying out the tests are described, that is, the characteristics of the equipment used, the conditions of the execution environment and any peculiar choices in this regard, as well as the effects of the prefetching technique used by the processor. This then gives way to *IV. Results*, where the theoretical knowledge acquired is applied in the interpretation of the values obtained, using different graphic representations as support. Finally, in *V. Conclusions*, a brief summary of the study is presented, the deductions made from it, and possible lines of future research are indicated.

II. Test methodology

This section describes the problem proposed and the organization of the programs developed to address it.

A. Problem Description

The key factor in the implementation of the problem revolves around performing a floating point sum reduction operation on R elements of a vector of *doubles* A[] with size N, access to these being determined by a parameter D so that the references follow the sequence A[0], A[D], A[$2*D$], ..., A[$(R-1)*D$]. Another input parameter to consider is the number of different cache lines (L) that one reads when using the elements of the vector.

In total, 35 experiments are carried out, with 5 different values of D (to be chosen by the programmers, in the range of integers between 1 and 100) and 7 of L: $0.5*S1$, $1.5*S1$, $0.5*S2$, $0.75*S2$, $2*S2$, $4*S2$, $8*S2$, where S1 and S2 are the number of cache lines that fit in the L1 cache and L2 data respectively. Each of these experiments is repeated 10 times to measure the number of clock cycles required and, then, the median of these 10 tests is calculated, so that 35 values are finally obtained, one for each pair (D, L) .

Furthermore, in each of the 350 total tests the reduction operation is carried out 10 times (which will increase the uniformity in the timing of the results, but will also be a factor in their analysis). The reduction totals are stored in a vector S[] over which the average is calculated and from which all elements are printed, in order to prevent the compilation from ignoring operations or making changes that could alter the conditions of the tests.

The number of terms of each test, R, is a value determined by the choice of D and L. The details of its obtaining are referred to the subsection **??**.

On the other hand, the reference to the elements to be added of A[] is not direct, but occurs through a vector of indices, ind[].

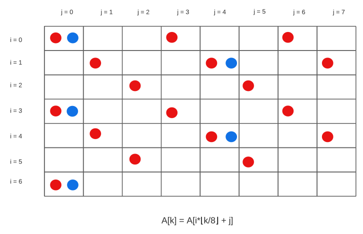
Note that the result of each test is not the number of clock cycles of the 10 associated reductions, but rather the number of cycles per access to each element. If we denote the total time in cycles as ck and the time in cycles per access as ck_access, we have:

$$ck\_access = \frac{ck}{10*R},$$

since for each reduction R elements are read from A[].

B. Getting R

Figure 1: *Graphic representation of the dependence of R on the range in which D is located for the test set: [1, 7] or [8, 100]. Using L=7, the accesses necessary when D=3 are shown in red; in blue, when D=12.*



To calculate the value of the parameter R in a given experiment (that is, with both (D, L) fixed), the following formula has been used:

$$R = \begin{cases} \left\lceil \frac{B*(L-1)}{D*\mathit{sizeof(double)}} \right\rceil & D < \frac{B}{\mathit{sizeof(double)}\,)} \\ L & D \geq \frac{B}{\mathit{sizeof(double)}} \end{cases}$$

This is due to the effect of the space between the elements of A being less or greater than the number of *doubles* that fit in a cache line, $\frac{B}{\mathit{sizeof(double)}}$. On the computer on which the tests have been carried out, $\frac{B}{\mathit{sizeof(double)}} = \frac{64\text{ bytes}}{8\text{ bytes}} = 8$ (see subsection A of *III. Execution*).

Suppose that the elements of vector A[] are found in consecutive cache lines, one after the other, as indicated in figure 1.

If D >8, there may be entire lines of A[] that are not accessed. On the other hand, in those that are accessed, only one element will be read. Since the number of lines accessed is by definition L, this will be exactly the value of R, without any influence from D.

Similarly, if D =8, only one item on each line will be accessed. Therefore, one should take as many terms as there are lines: $R = L$.

However, if D <8, more than one piece of data can be selected per line. The space should be divided into blocks of length D. Nonetheless, this length may not be an exact divider of 8. Consequently, the ceiling() function will have to be used to ensure that the "line spacing" blocks are also included. In fact, it can be further refined with respect to the last line, since it only takes one value to determine whether the line is accessed. Hence, it is only multiplied by L-1 inside ceiling() and added by 1 later.

C. Getting N

The total size of the array, N, will be calculated as a function of R and D:

$$N = D * (R - 1) + 1$$

Again, one can visualize its meaning by thinking about blocks of D elements. There will be an R number of these blocks, except from the fact that in the last of them only one element needs to be included (the first one, which will be the R-nth, since none of the following ones will need to be read). Consequently, an array of size $D * (R - 1)$ will be defined with an extra element.

D. Program Structure

The structure of the program has been divided into 4 components:

- **reduction.c**, the main program. Its purpose is to conduct an experiment for which the values of D and L have been fixed, which are introduced as arguments. After running the reduction, measuring its cycles, and calculating the time per access, it writes the results to a temporary file. The timekeeping functions can be found in the *counter.h* library and were provided as course material.
- **median.c**, a program to filter the results. It acts on 10 measurements obtained from *reduction.c* using the same values of D and L. The goal is to calculate the median of these measures, removing possible outliers. Likewise, it is possible to centralize the results in a single piece of data for each pair (D, L). These values are written to a final results file.
- **bash_script.sh**, a bash script that automates execution. After manually preselecting the 5 constants to be used as D, the 7 L constants specified in the study are iterated on. For each pair (D, L), the *reduction* program is called 10 times. After the $5 * 7 * 10$ =350 tests, the registration of the temporary file will be complete. At that point, *median* is called to get the final results.
- **graphics. R**, a data visualization R script. Through it, different measures and relationships of the program are represented in the form of 3 2D graphs and 2 3D graphs, with the purpose of helping and supporting the analysis and understanding of the study.

The following are the most significant characteristics of each of the components referred to:

D.1 reduction.c

First of all, it should be noted that the dynamic memory allocation has been made with the function _mm_malloc() to align the beginning of the vector with the beginning of the cache line and ensure the correct study of the locality.

The vector A[] is initialized with random values in the set $(-2, -1] \bigcup [1, 2)$. In this way, one can be sure that before starting the real tests there is already data previously loaded in the different levels of the memory hierarchy (cache, TLBs, etc.), avoiding possible alterations in the measurements due to these being empty and requiring additional actions. Thus it is ensured that each line will be brought in the same conditions: in the tests there will be no mandatory misses ([2], [3]), since each and every one of the elements of A[] will have already been referenced beforehand. Only capacity and conflict misses are considered.

Since in most computers the line size is invariant between caches, it has been operated under the assumption that this is generally true. Otherwise, note that a refinement of the parameter calculation would be necessary.

A function has been implemented that aims to disarrange the elements of the vector ind[], which stores the indices of vector A[] corresponding to the elements to be used in the reduction. The full justification and analysis of the results can be found in IV. Importantly, to ensure that the ind[] vector is actually random, the decluttering process is repeated until it satisfies the supported standards (it is compared to the ordered vector and re-ordered if the match percentage is not less than 15%). The randomization code was developed by Ben Plaff ([7]), and is a refinement of the Fisher-Yates algorithm.

D.2 median.c

The median has been chosen as a measure of data centralization. The main reason is that it is robust (i.e., that it is affected to a low extent by outliers) compared to other estimators such as the mean. In this way, the effect of interference that other processes may have, peaks or drops in the frequency of the processor, etc., is minimized. Applying the median ensures that the value obtained for certain initial conditions has a high degree of reliability.

Since the number of executions for each pair (D, L) is even (10 measurements), a correction had to be applied to the calculation. We have chosen to interpolate the two central values and then use the average of both. This is still a fairly safe and statistically standard option, because having previously sorted the data continues to discard the extreme results.

In this program, the insertion algorithm is used to sort the data. The choice is to optimize the execution time taking into account that we are working with small arrays ([8]).

D.3 bash_script.sh

The 10 executions for the same pair (D, L) are not performed consecutively, but are interleaved with other values. The explanation of this decision can be found in the section F.

D.4 graphics. R

We have chosen to use R as the language for the representation of the data because of its wide range of possibilities for visualization. The code uses the *plotly*, *plot3D* and *rgl* graphic libraries, which must be installed in advance.

III. Execution

This section specifies the technical characteristics of the processor and the memory hierarchy of the computer used for the execution of the code associated with the practice, discusses the conditions imposed on the environment outside the program during the experiments, and provides a comment on prefetching both theoretically and with respect to the processor in question.

A. Technical data of measuring equipment

As far as the hardware used is concerned, the tests have been carried out on a Lenovo V130-15IKB with the following specifications:

- Processor: Intel® Core™ i7-7500U quad-core 2.70 GHz.
- Operating System: Ubuntu 20.04.4 LTS.
- Operating System Type: 64 bits.
- Kernel: 5.13.0-35-generic.

It should be noted that this is a laptop, and that its processor is 7th generation and low power (U), that is, its design puts battery life before general performance.

The different attributes of the computer's memory hierarchy have also been checked. It has a RAM of 7.1 GiB and 3 cache levels, all of which are fully associative. There are several ways to check its technical details, such as through the Windows Task Manager, with the CPU-Z program, etc. We have chosen to use the sysconf() function in a C program specially dedicated to this purpose (*info_cache.c*, which is included as the **??** section of the appendix).

sysconf() allows us to check the number of ways (for example, for L2, with _SC_LEVEL2_caché_ASSOC), the total byte size (_SC_LEVEL2_caché_SIZE), and the line size in bytes (_SC_LEVEL2_caché_LINESIZE). Thus, using that the number of lines (S1, S2) will be equal to the total size in bytes divided by the line size in bytes, we have:

Table 1: *Características técnicas de la caché del equipo y consecuentes valores de L.*
L1 I y L1 D corresponden a las cachés L1 de instrucciones y datos, respectivamente.
S_i representa el número de lineas caché de cada nivel.

caché	Tamaño	Tamaño de línea
L1 I	2^{15} bytes	64 bytes
L1 D	2^{15} bytes	64 bytes
L2	2^{18} bytes	64 bytes
L3	2^{22} bytes	64 bytes
caché	S_i	**Tipo de caché**
L1 I	512 = 2^9	Asociativa por conjuntos
L1 D	512 = 2^9	Asociativa por conjuntos
L2	4096 = 2^{12}	Asociativa por conjuntos
L3	65536 = 2^{16}	Asociativa por conjuntos
caché	**Nº de vías**	**Nº de conjuntos**
L1 I	8	64
L1 D	8	64
L2	4	1024
L3	16	4096

$$B = \text{sysconf(\_SC\_LEVEL1\_Dcache\_LINESIZE)}$$

or

$$B = \text{sysconf(\_SC\_LEVEL2\_cache\_LINESIZE)}\ ,$$

and

$$S1 = \text{sysconf(\_SC\_LEVEL1\_Dcache\_SIZE)} * \frac{1}{B}$$
$$S2 = \text{sysconf(\_SC\_LEVEL2\_cache\_SIZE)} * \frac{1}{B}$$

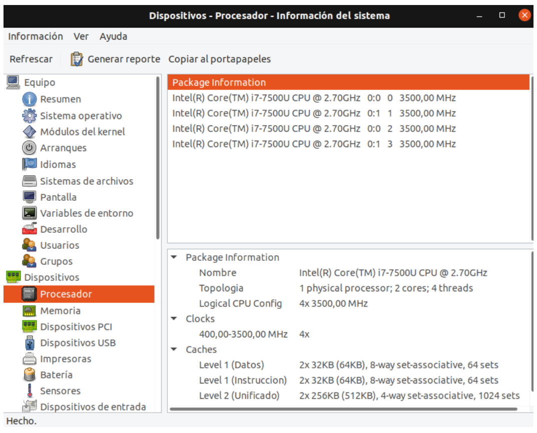
One can see the results obtained in the 1 box. Note that the following ratio is used to calculate the number of sets:

$$\text{No. of sets} = \frac{\text{No. of lines}}{\text{No. of ways}} = \\ = \frac{\text{Total size}}{\text{Line size * No. of ways}}$$

Based on the data in the 1 box, the values of L to be used in the program are:

- 0.5 * S1 = 0.5 * 512 = 256
- 1.5 * S1 = 1.5 * 512 = 768
- 0.5 * S2 = 0.5 * 4096 = 2048
- 0.75 * S2 = 0.75 * 4096 = 3072
- 2 * S2 = 2 * 4096 = 8192
- 4 * S2 = 4 * 4096 = 16384
- 8 * S2 = 8 * 4096 = 32768

Figure 2: *Cache level information obtained with hardinfo. Note the multiplicity of L1 levels of instructions and data and L2 level.*



```
Dispositivos - Procesador - Información del sistema

Package Information
Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz  0:0  0  3500,00 MHz
Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz  0:1  1  3500,00 MHz
Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz  0:0  2  3500,00 MHz
Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz  0:1  3  3500,00 MHz

▾ Package Information
    Nombre              Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz
    Topologia           1 physical processor; 2 cores; 4 threads
    Logical CPU Config  4x 3500,00 MHz
▾ Clocks
    400,00-3500,00 MHz  4x
▾ Caches
    Level 1 (Datos)        2x 32KB (64KB), 8-way set-associative, 64 sets
    Level 1 (Instruccion)  2x 32KB (64KB), 8-way set-associative, 64 sets
    Level 2 (Unificado)    2x 256KB (512KB), 4-way set-associative, 1024 sets

Hecho.
```

As an additional detail, it should be noted that in this computer each of the L1 and L2 caches are shared between 2 cores. Due to the peculiarity of this fact with respect to the usual structure of memory hierarchies, it has been verified both through the information available in the *sys/devices/system/cpu/* directory of Linux, and through *hardinfo*: see figure 2.

B. Execution Conditions

All executions of the *bash_script.sh* script have been carried out with as few running processes as possible (a process in the shell to be able to launch the program, and those that are essential for the operating system). This is a precautionary measure that seeks to minimize the impact of background tasks, since experiments are sensitive to interruptions, memory use by other processes, etc.

Also, to avoid any kind of interference between tests resulting from the decisions of the operating system scheduler, none of them is parallelized.

Each test has also been run in a new process, instead of running the same program multiple times, to force all experiments to operate under initial conditions that are as similar as possible and that lines from previous runs were not reused, but had to be loaded from scratch.

On the other hand, it has operated under two different sets of compilation constraints:

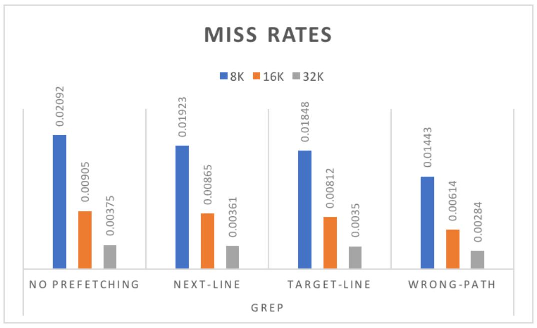
First, as practice required, the gcc compiler with the -O0 option was used. It avoids certain compiler optimizations, such as software prefetching, not including functions without side effects in the assembly code, unwinding loops (which optimizes loop execution by removing or reducing certain instructions), the use of vector instructions, etc. ([**?**]). Thus, it is intended that the process of translating the C code into assembly language maintains a high fidelity with respect to the original, so that users can analyze the internal behavior of the computer based on the programmed instructions.

Compilation with the -O0 option has always been accompanied by a process of randomizing the ind[] index vector. Therefore, the associated executions seek to minimize the locality of the program as much as possible.

Second, we have chosen to use the -O3 option and not clutter the array of indexes. This choice has the opposite purpose to the previous one: to maximize the optimizations and efficiency derived from the locality, with the aim of being able to contrast the results with the former. With the -O3 option, one will be allowing the unwinding of loops in a wide range of situations, allow the compiler to replace function calls with the function code, and as well not prevent 'aggressive' optimizations in the compilation.

C. Prefetching

Figure 3: *Benchmark failure rate of the grep command by applying different prefetching algorithms. Technical details can be found at [9].*



Prefetching is a cache failure reduction technique that predicts future requests for data, which are cached before the processor actually requires them ([1]). In this way, if the predictions are good (which is closely related to the locality principle), the processor will have the data it requests available in cache as it executes the program, without suffering a penalty in terms of time for having to search for it in main memory. An example of the impact of this refinement can be seen in the figure 3.

The basic principle of prefetching is based on loading from main memory to the cache not only the requested data, but also these and some close or that have been determined, by prediction heuristics, with high probabilities of being required in the near future.

A distinction can be made between software prefetching and hardware prefetch. The first is compiler-controlled and uses techniques such as *loop unrolling* to avoid data risks for jumps in pipelining. However, it's important to balance the positive effects of prefetching with the *overhead* of its additional operations, so that it ends up being a performance improvement rather than a detriment.

On the other hand, hardware prefetching is the responsibility of the processor itself. Usually, as is the case with the L1 and L2 levels of the Intel Core i7 (precisely the processor of the test team), this process is done by adding the cache line immediately after the requested one to the cache buffer ([1]). It will be reasoned in section G why, based on the results obtained, this is the criterion that is almost certainly being used.

Actually, the ideal execution situation would be to eliminate the prefetching entirely. The -O0 build option allows us to avoid software optimizations, but it is not possible to circumvent the impact that hardware *prefetching* will have. This is simply one more factor to take into account when analysing the results.

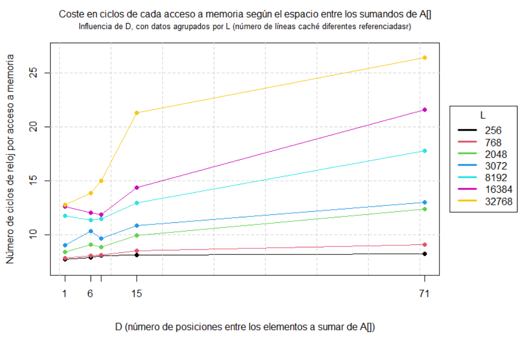
IV. Results

This section presents, interprets and justifies the different results obtained in practice. To do this, different graphs built in R and a heat map obtained through Excel will be included.

This section is structured in different subsections that refer to conditions on the results and aspects of interest in them.

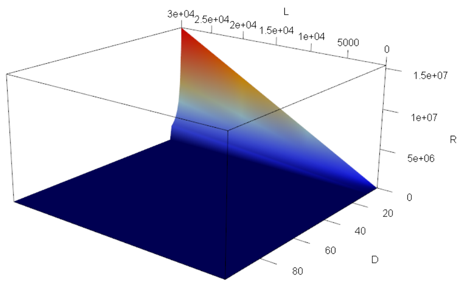
A. Choice of D

Figure 4: *Graphical representation of access times for the values of D 1, 6, 8, 15, and 71 with the -O0 execution option and with the ind[] array out of order.*



One of the clearest observations is that access times get worse as D grows, as can be seen in Figure 4. The explanation lies in the very nature of D: if it increases, elements will be requested further and further away from each other, so that the locality decreases significantly.

Figure 5: *Graphical representation of the get function of R (Z-axis) with D (Y-axis) and L (X-axis) as inputs.*

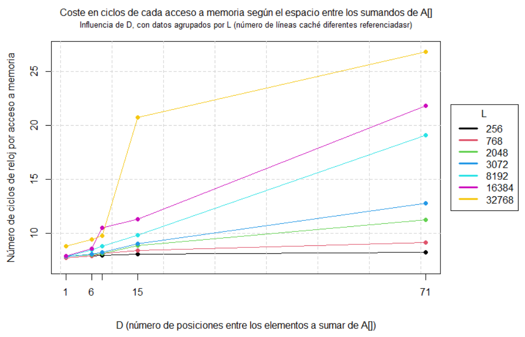


When you access elements in an orderly manner and $D = 1$, you have a sequential locality situation (a particular case of spatial locality), since you are reading consecutive elements. In addition, each reduction operation will have a temporal locality with respect to the same cache line, since $\frac{B}{sizeof(double)} = \frac{64}{8} = 8$ in a row will be used each line.

In the case of -O3 and non-randomized ind[] runs, it is observed that in this situation very low access times are obtained for any value of L. Since data movements in the cache hierarchy are performed in bulk (as full lines), for each failure there would be 7 subsequent hits, and consequently the performance is high.

If it is run with -O0 and we randomize ind[], the results are significantly worse: we go from an average of 7 seconds (almost independently of L) to more than 14 seconds for the highest value of L. This is to be expected, because with the latter option you are no longer accessing the data sequentially and, in addition, you have a very high number of terms (N), since D is inversely proportional to R (see the graph **??**), so that you end up needing to replace a large number of lines, especially for large L.

Figure 6: *Graphical representation of access times for the values of D 1, 6, 8, 15, and 71 with the -O3 execution option and with the ordered ind[] array.*



When $1 < D < 8$, there is hardly any deterioration in performance. In the case of -O0, it is observed that there is a difference of between 1 and 2 seconds for most values of L with respect to $D = 1$. However, the times are still low and do not exceed 15 seconds. The rationale is that although fewer items are now accessed per cached line, lines are reused multiple times in each reduction. When L is small, this is especially significant, as lines are less likely to be replaced and therefore the data will still be cached. The case in which ind[] is ordered is even better (note the figure 6), even without taking into account the rest of the -O3 optimizations, since the line will almost certainly be available for all the accesses that are required of it, as it occurs in an orderly manner. When $8 \leq D$ <16, the formula for R changes: R now equals L. That is, from each line only one element is taken at most (if $D = 8$ exactly one is taken, if $D > 8$ there may be lines of A[] that are not used). At this point, the effects of prefetching begin to be felt. Previously, it was certain that the index summand immediately following a given one would be on the same or next cached line. Taking for granted the assumption that the *prefetching* hardware brings two lines, the required one and the one that succeeds it, this was a high improvement in performance. Even in the case where ind[] is unsorted, when L is small and there are no capacity cache misses ([2], [3]), as in the case $L = 0.5 * S1$, or these are very scarce, as in the case $L = 1.5 * S1$, the required cache lines will be available for use because they have been brought in earlier.

Now that there are lines that are no longer used, this is no longer true, and the magnitude of the benefits of prefetching decreases. In fact, one of the arguments that support the hypothesis of the extra line by hardware prefetching is the notable worsening in the results when $D \geq 16$, since in this case the extra line is never used. The decrease in performance occurs for both the -O0 and -O3 options, but the former is the truly illuminating one, as the software prefetching of the latter prevents us from precisely defining the causes of the differences found.

From this value onwards, as D increases, the results continue to worsen. An almost linear behavior can be seen with a slope that becomes more acute as L increases. In fact, there are hardly any differences between the two types of execution when L is very high, and in this sample both reach a maximum value of around 26 seconds.

From this value onwards, as D increases, the results continue to worsen. An almost linear behavior can be seen with a slope that becomes more acute as L increases. In fact, there are hardly any differences between the two types of execution when L is very high, and in this sample both reach a maximum value of around 26 seconds.

Based on the above explanations and various executions, it is estimated that the most significant conclusions emerge from choosing 5 D values in lower ranges. The set $\{1, 6, 8, 15, 71\}$ has been considered to be a representative sample of the peculiarities explained, and is reflected in the figures in this section [1].

B. Locality

The most important locality ([**?**]) in this study is, without a doubt, the spatial one. The temporal locality on the same piece of data only takes effect 10 times, once by reduction, since

[1]Although only representations of the set 1, 6, 8, 15 and 71 are shown in this document as values of D, it has also been considered of interest to use 1, 3, 7, 8 and 16; 10, 30, 50, 70 and 90; and 14, 15, 16, 17 and 18. Its associated diagrams in the two modes of execution described are included as an appendix, and it is recommended that they be checked in order to visualize in more detail the details described in this study.

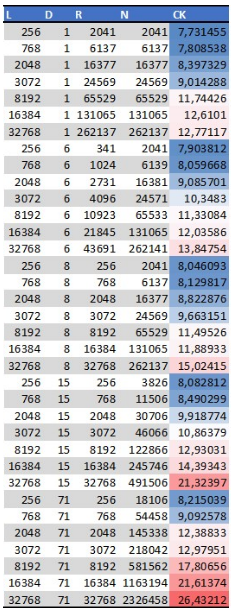


Figure 7: *Heat map of the 35 results obtained with the -O0 run option and the ordered ind[] array.*

L	D	R	N	CK
256	1	2041	2041	7,731455
768	1	6137	6137	7,808538
2048	1	16377	16377	8,397329
3072	1	24569	24569	9,014288
8192	1	65529	65529	11,74426
16384	1	131065	131065	12,6101
32768	1	262137	262137	12,77117
256	6	341	2041	7,903812
768	6	1024	6139	8,059668
2048	6	2731	16381	9,085701
3072	6	4096	24571	10,3483
8192	6	10923	65533	11,33084
16384	6	21845	131065	12,03586
32768	6	43691	262141	13,84754
256	8	256	2041	8,046093
768	8	768	6137	8,129817
2048	8	2048	16377	8,822876
3072	8	3072	24569	9,663151
8192	8	8192	65529	11,49526
16384	8	16384	131065	11,88933
32768	8	32768	262137	15,02415
256	15	256	3826	8,082812
768	15	768	11506	8,490299
2048	15	2048	30706	9,918774
3072	15	3072	46066	10,86379
8192	15	8192	122866	12,93031
16384	15	16384	245746	14,39343
32768	15	32768	491506	21,32397
256	71	256	18106	8,215039
768	71	768	54458	9,092578
2048	71	2048	145338	12,38833
3072	71	3072	218042	12,97951
8192	71	8192	581562	17,80656
16384	71	16384	1163194	21,61374
32768	71	32768	2326458	26,43212

each term is used only once in each operation. Even if we consider temporal locality on each cache line as a whole, it will only take effect on a single drawdown when D <8.

In contrast, D controls spatial locality directly. The smaller it is, the more this principle is fulfilled, as nearby array data will be required. If L is small, which implies that the probability of cache line replacement is not too high, this fact can be used even in the case of deordering ind[] so as not to suffer the penalty of going to main memory, thanks to *prefetching*. In the case of using hardware prefetch, this effect will only be exploited if D is less than 16, but with software prefetching the algorithms could be adapted to more varied sizes.

This analysis can be carried out, for example, through the heat map in Figure 7.

C. The case for D=1

When using the -O0 option, it is relatively common to find that smaller values of D (such as 1 or 3) perform worse than somewhat higher choices, such as D =6 or D =7, especially when L takes high values. As a result, an initial downward trend can be observed in the representations of access times as a function of D.

The explanation is found in the expression of the formula for R when $D < 8$ (B). The graph of the function (5) gives away the reason: you can see that R triggers when D is small and L is large, since R could be approximated as $R \approx 8 * (L - 1)$. Looking also at the formula for N, $N = D * (R - 1) + 1$, it is also not unreasonable to assume $R \approx N$.

Consequently, the number of terms is exaggeratedly high, reaching levels of the order of 131065 for (D=1, L=16384), or 262137, for (D=1, L=32768). In the long run, this will lead to a high number of cache replacements due to capacity misses and conflicts. It also reduces the chance of being able to reuse a line between different reductions in the group of 10 per test.

When ind[] is ordered, the consequences are not as severe, as a sequential or quasi-sequential locality is used, but if ind[] is disordered, the effects are catastrophic, hence the peaks in the -O0 graphs.

When we increase D slightly, until we reach for example D=6, R no longer exceeds the range of 50000 for any value of L. In this case, the number of terms is acceptable and the asymptote of the function of R at $D = 0$ ceases to have so much effect.

D. L and R values

It is observed that an increase in the value of L also causes a greater number of access cycles, both in both -O0 and -O3 executions. This subsection will deduce, based on the theoretical properties of the memory hierarchy, why the number of capacity and conflict misses is dependent on L

When L is small, most of A[]'s cache lines will fit in the cache tiers closest to the CPUs. Since the only data used in the 10 reductions is the A[] vector, two iteration variables (i and j), and the result vector S[], the cache lines used can be hosted in the L1 data cache without capacity misses and conflicts. In particular, this is what happens when $L = 0.5 * S1$.

If L grows, the number of cache lines that will need to be used in each reduction cannot be fully stored at the L1 data level. Therefore, it will be necessary to make movements from lower levels. Specifically, at least the L2 cache will be used, since every piece of data must be available in all lower layers.

This incurs a penalty for failure, as a line has to be substituted and the access speed is lower the further down the hierarchy is lower ([2]). The impact on the associated access time is direct.

In addition, on this computer, the L1 cache is the only one that is divided between data and instructions. Using L2 and L3 levels means that the data will be in direct competition for space with the instructions, which leads to an increase in the number of conflict misses. Despite this, in the case of this code this is not a very worrying factor, since it uses a loop of few instructions that will end up being completely contained in the L1 of instructions.

What does affect, however, is associativity: it was mentioned in the box 1 that the L2 cache has only 4 ways, while each L1 has 8. This lower number of lines available per set results in an increase in the number of conflict misses.

The theoretical basis described is consistent with the specifications of the problem and with the results. L directly affects the value of R, whether it $D < 8$ or $D \geq 8$, in the first case in a cushioned way and in the second directly. An increase in R implies the need to move more lines between levels of the memory hierarchy (and thus the occurrence of more capacity misses).

Additionally, if ind[] is out of order, there is not even an acceptable probability that a line can be used more than once among the 10 reductions, since a high L causes so many misses that by that time it is likely that the line in question has already been replaced.

The interpretation described here is consistent with research in the field such as [6].

E. Size of A[]

The value of N is also significant, with special emphasis on cases where we have software prefetching and ind[] is sorted. As the size of the array increases, there will be more reference data against which to base prefetching predictions, so that it will eventually become more effective (if ind[] is disordered, the impact would be reduced or disappear entirely).

Figure 8: *Graphical representation of access times for each value of L, 0.5*S1, 1.5*S1, 0.5*S2, 0.75*S2, 2*S2, 4*S2 and 8*S, with the -O0 execution option and with the ordered ind[] array.*

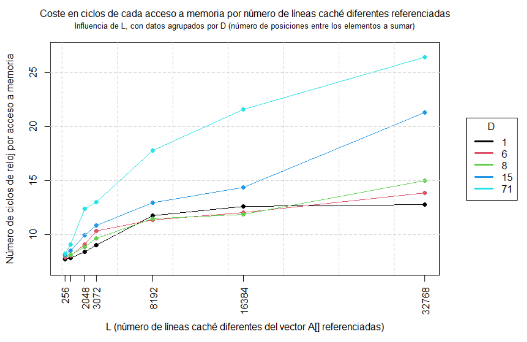
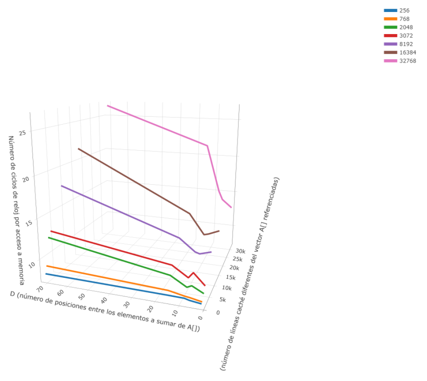


Figure 9: *3D graphical representation of the times (Z-axis) for each value of L (X-axis) and D (Y-axis), with the -O0 execution option and with the ordered ind[] array.*



Now, since $N = D * (R - 1) + 1$, if we want to maximize N we will have to have a large value of D (or at least greater than 7, so that R is equal to L and not inversely proportional to D) and a large value of L. This is why the slopes of -O3 graphs are less steep than those of -O0 in high D ranges.

F. Execution Order

At the start of the testing process, we started by testing a bash script that ran the tests using the following loop:

```
for each L ∈ listL
    for each D ∈ listD
        for k=0..9
            do test
```

However, there were numerous irregularities in the results between tests. It was then hypothesized that the order of the loop might be significant for the measurements. Indeed, it was found that changing the order to:

```
para k=0..9
    for each D ∈ listD
        for each L ∈ listL
            do test ,
```

the graphs became much more uniform. It was deduced that the main reason was an incorrect assumption of independence between the tests: since initially the 10 repetitions of each experiment (a fixed pair (L, D)) were launched consecutively, any background processes that the team was running at the time to the developer's dismay would end up affecting several of the tests. so that the majority would be displaced and, consequently, so would the median. Once the tests are distributed and further away over time, the likelihood of a child process affecting several of them significantly is minimized.

G. Hardware prefetching

As discussed in A, hardware prefetching with an extra line is responsible for much of the difference in timing when D moves from range [1, 15] to range [16, 100]. At this point, the locality is so low that the *overhead* of the additional operations it entails ends up superimposing its effects on efficiency, which disappear if we consider the hypothesis about the number of lines brought by prediction to be true, which is quite likely based on the results.

The most obvious consequence in the visualizations is the sharpening of the slopes of access times when the threshold of $D = 16$ is exceeded. From that point on, the increase is practically linear.

For some time it was hypothesized that the passage of D from one multiple of 8 to another would significantly affect the measurements. However, no statistically significant evidence was found and the idea was eventually discarded. The only multiples of 8 with real consequences are 8 and 16, for the reasons stated above.

Figure 10: *Graphical representation of access times for the R/L ratio, representative of the locality (especially the spatial locality), with the -O0 execution option and with the ordered ind[] array.*

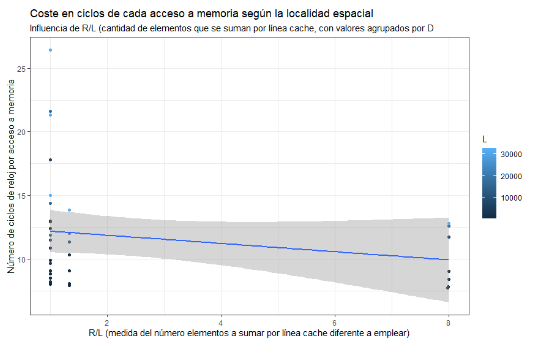
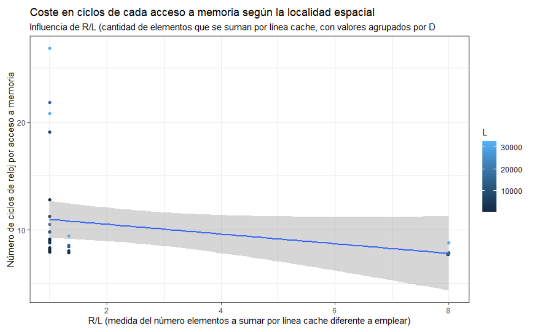


Figure 11: *Graphical representation of access times for the R/L ratio, representative of the locality (especially the spatial locality), with the -O0 execution option and with the ordered ind[] array.*



H. R/L ratio

The R/L ratio can be understood as an estimate of the locality of the program: it represents the number of terms that are taken per required cache line. A representation has been developed in this regard that demonstrates the inversely proportional relationship between locality and access times. Note that the high ratio of dots for $R/L = 1$ is due to $R = L$ for $D \geq 8$, so the really interesting values in this graph are in the range $R/L > 1$.

Although R/L is indicative of locality in general, it is especially interesting to interpret it through spatial locality in particular, as it is the criterion on which it has a direct and palpable effect.

The graphs 10 and 11 show a simple regression line that approximates the relationship between locality and time per access under the assumption of linear character. A confidence interval for line location is also included.

V. Conclusions

This report has studied a major issue when it comes to reducing latency in microprocessors, an aspect that has recently been addressed from the point of view of the use of the cache [10], [11], providing a complementary view. Thus, the effect of spatial and temporal locality, memory architecture and other factors (prefetch, number of data access, independence of execution tests, etc.) on the efficiency of memory access times has been analyzed.

The requirements of the problem in question, the technical details of the hardware used and the theoretical basis of the points mentioned have been used as the basis for the analysis of the results. These reflect a strong dependence of access times on locality, as well as the percentage of utilization of each cache level, two closely related factors. A low locality (high D) results in a large number of data movements between different levels of the hierarchy, and if the amount of data itself is high (L and R of considerable magnitude), conflict and capacity misses cause significant penalties in access times.

Likewise, the relevance of the control of independence in the test environment and smaller factors such as the size of the data structures (which facilitates prefetching and increases R) or the ratio between R and L (indicative of the locality) have been observed.

Finally, the type of hardware prefetching implemented on the testing computer has been critical to understanding locality-based predictions (see A section of IV. Results). Therefore, the study of the specifications of prefetching, especially the reference of consecutive cache lines, is considered to be a matter of interest for future studies, as well as the execution of the program in other architectures to analyze the effects of memory hierarchies (cache types, sharing between cores, and distinctions between data and instructions).

References

[1] Hennessy, John L. et al. (5ª Edición). Arquitectura de Computadores: Un enfoque cuantitativo. Cap. 2: Diseño de la jerarquía de memoria. (pp. 78-105). Editorial Morgan Kaufmann, Elsevier. *Intel virtualization technology.* Computer, vol. 38, no. 5, pp. 48–56, 2005.

[2] Patterson, David A. y Hennessy, John L. (5ª Edición). Computer Organization and Design: MIPS Edition. Chapter 5: Large and Fast: Exploiting Memory Hierarchy. (pp 372-499). Editorial Morgan Kaufmann, Elsevier. 2014.

[3] Brihadiswaran, Gunavaran. Taxonomy of caché Misses. *The High Performance Computer Forum.* 2020. `shorturl.at/ehET9`, [online] última visita 23 de marzo de 2022.

[4] Harris, Sarah L. y Harris, David Money. Digital Design and Computer Architecture. 8 - Memory Systems. (pp 486-529). Editorial Morgan Kaufmann. 2016.

[5] Selecting Optimization Options. *Arm Keil, Compiling Getting Started Guide*. Versión 6.16. `https://www.keil.com/support/man/docs/armclang_intro/armclang_intro_fnb1472741490155.htm` [online] última visita 22 de marzo de 2022.

[6] Morris, Gerald R. The effect of caché on memory access time. *ERDC DSRC, DoD Supercomputing Resource Center.* 2003. `https://www.erdc.hpc.mil/docs/Tips/cachÃĬ20030711.pdf`, [online] última visita 23 de marzo de 2022.

[7] Plaff, Ben. How can I shuffle the contents of an array? *Ben Plaff* 2004 `https://benpfaff.org/writings/clc/shuffle.html`, [online] última visita 20 de marzo de 2022.

[8] Tang, Daisy. CS241 – Lecture Notes: Sorting Algorithm *CalPolyPomona, CS241: Data Structures and Algorithms II* Referencia del curso: Carrano, Frank M. Data Structures and Abstractions with Java. 4th Edition,

2014. `https://www.cpp.edu/~ftang/courses/CS241/notes/sorting.htm`, [online] última visita 20 de marzo de 2022.

[9] Alsultaan, Heba et al. Final Report Implementing and Testing Four Prefetching Cache Algorithms. *University of Colorado Denver. Advance Computer Architecture.* 2017 `https://www.researchgate.net/publication/328597173_Final_Report_Implementing_and_Testing_Four_Prefetching_Cache_Algorithms` [online] última visita 23 de marzo de 2022.

[10] Perarnau, Swann Tchiboukdjian, Marc Huard, Guillaume. (2011). Controlling Cache Utilization of HPC Applications. Proceedings of the International Conference on Supercomputing. 295-304. 10.1145/1995896.1995942.

[11] Mobile Edge Cache Strategy Based on Neural Collaborative Filtering - Scientific Figure on ResearchGate. Performance comparison of content cache space utilization of different algorithms.