Examination of the Effects of Order on Array Accessing in Two Functionally Equivalent C Routines using MIPS

Sara Huang

March 9, 2018

Abstract

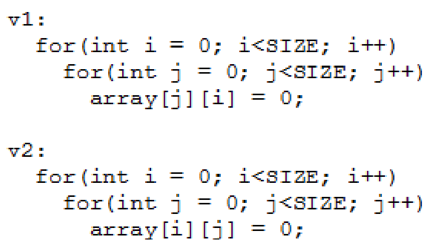
Most C programs have some sort of array accessing in their routines, however they are not always the most efficient and cannot always generate the best performance for a given data set. We introduce a translated MIPS assembly code analysis using a proper benchmark bashscript on realtime as a measure for efficiency, and experiment with differently sized arrays. Further, in our study of the performance disparity between functionally equivalent C routines, we explore the ramifications and implications of memory on computer architecture.

1 Introduction

Array accessing in general is a decision rule between 'read' and 'write' to elements and parts of arrays. It is most often used to find out specifically which elements are being accessed by a given source code. Given an array, it may be traversed and initialized for each value to some integer in the process. An array, in data structures and algorithms, is a fixed-size, finite, and sequential storage of elements, each having adjacent memory locations that correspond to each address. For example, the lowest address would correspond to the first element in the array and the highest address would correspond to the last element in the array. The memory is where the programs are stored when they are run and it contains the information needed by the running programs[1]. Methods that utilize memory allocation and how well the computer can handle workload are usually tested against a benchmark that chooses to measure performance in particular. A benchmark sets a standard for comparing programs and is the norm for meeting specifications.

2 Problem Statement and Analysis

Our array accessing task is to discern between two seemingly identical C routines and explain the performance discrepancy between them. Our goal then, therefore, is to analyze the two source codes, determine the differences, and investigate the performance metrics of each C program, as shown in Figure 1. Let us restrict our study of arrays to be two-dimensional and uninitialized in C, so from now on, we refer to an 'array' as such. Using this new term, we would like to know if the order matters when initializing the elements in the array. Generally, accessing an element of an array takes constant time in both the worst case and the average case. A common misconception with this is that the order does not matter, but it does when executed on a computer. In the present work, we explore the reasons for this using a proper benchmark for the C programs and statistical analysis of experimental data.

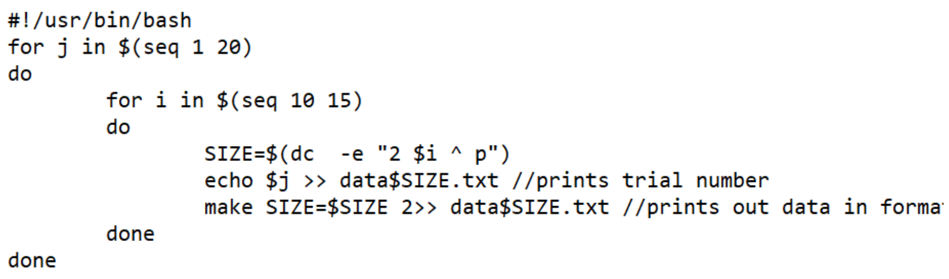


```
v1:
  for(int i = 0; i<SIZE; i++)
    for(int j = 0; j<SIZE; j++)
      array[j][i] = 0;

v2:
  for(int i = 0; i<SIZE; i++)
    for(int j = 0; j<SIZE; j++)
      array[i][j] = 0;
```

Figure 1: Portions of the original C program codes looping through the arrays v1 and v2 by columns first and then rows, or rows first and then columns, respectively.

Let us limit our discussion to the following C program in Figure 1 that has outputs of v1 and v2 as arrays, and i and j as the row and column number, respectively. For both v1 and v2, the initialization starts with i = 0 and goes up from there up to the given size. The similarities end there. Now, for v1, the array is accessed in such a way that v1 goes by columns and then by rows and for v2, the array is accessed in such a way that v2 goes by rows and then by columns. The difference, albeit subtle, drastically changes the runtime of each array. An explanation for this is that since the rows are loaded first in memory, all the elements in the rows will be loaded before proceeding to the next column to be iterated. This makes the access process much faster as less time would be spent retrieving all the elements per row before the columns, following the way that memory works.



```
#!/usr/bin/bash
for j in $(seq 1 20)
do
        for i in $(seq 10 15)
        do
                SIZE=$(dc  -e "2 $i ^ p")
                echo $j >> data$SIZE.txt //prints trial number
                make SIZE=$SIZE 2>> data$SIZE.txt //prints out data in forma
        done
done
```

Figure 2: The bashscript code looping through twenty trials for powers of 2 starting from ten and ending at fifteen.

For the benchmark parameters, we choose to use realtime because we can account for all possible circumstances that contribute to the program's overall run[2]. Bias results from using CPU time or user time because they only focus on the execution of the program and not of the program in its entirety. Figure 2 shows the following bashscript code that was used to simulate different square and non-square array sizes with respect to powers of two. Twenty trials each were performed with data extraction on differently-sized arrays of increasing powers of 2 starting from 10 and ending with 15.

In order to better explain why there exists a stark difference in realtime between the two programs, conversion into MIPS assembly code is used to highlight the reasons. Figure 3 shows the translation in detail. At first glance, the two may look the same again but it is in the different assignments of the registers as rows or columns that makes all the difference. The contrast lies in the order of which the array is implemented[2]. For v1, the order takes the height of the array first and shifts by 2 (in essence, multiplying by 4) to store words into it. For v2, the order takes the width of the array first and also shifts by 2 to store words into it. The latter is much faster since the load-immediate takes in the values of the rows first which go linearly across. In the former, the load-immediate takes in the values of the columns in the array first, before going through the rows, which delays the loading into memory as the initialization needs to happen for all the rows in order to get the column values. The storage of the contents in memory are arranged by rows first and then columns so it would be much faster to traverse through rows or width first and then by height. This is responsible for the realtimes of each program.

```
v1
          li $t0,SIZE          #outer loop
          li $t1,0
v1loop:   beq $t1,$t0,exit     #if equal, go to exit
          li $t2,SIZE          #load for inner loop
          li $t3,0
loopv1:   beq $t2,$t3,v1loop   #if equal, go to v1loop
          li $t4,array         #load array
          move $t5,$t3
          add $t5,$t5,$t4      #height of array
          sll $t5,$t5,2        #shift left 2 (x4) for registers
          lw $t6,0($t5)        #want value
          move $t5,$t1
          add $t5,$t5,$t6
          sll $t5,$t5,2
          sw $t5,0($zero)      #store 0 in $t5
          addi $t3,$t3,1
          j loopv1
          addi $t1,$t1,1
          j v1loop

exit:


v2
          li $t2,SIZE          #outer loop
          li $t3,0
v2loop:   beq $t2,$t3,exit     #if equal, go to exit
          li $t0,SIZE          #load for inner loop
          li $t1,0
loopv2:   beq $t1,$t0,v2loop   #if equal, go to v1loop
          li $t4,array         #load array
          move $t5,$t3
          add $t5,$t5,$t4      #width of array
          sll $t5,$t5,2        #shift left 2 (x4) for registers
          lw $t6,0($t5)        #want value
          move $t5,$t1
          add $t5,$t5,$t6
          sll $t5,$t5,2
          sw $t5,0($zero)      #store 0 in $t5
          addi $t1,$t1,1
          j loopv2
          addi $t3,$t3,1
          j v2loop

exit:
```

Figure 3: The sketch of the MIPS assembly code conversion of the difference of the two C programs for arrays v1 and v2.

3 Results

Here we present our results for the proper benchmark realtime performance metric, as well as the varying sizes of arrays used in our simulations.

3.1 Square Arrays

For square arrays, the dimensions of the rows and the columns of the arrays are equal in value. Based on the benchmark bashscript, we performed a total of twenty trials per array size in powers of 2 starting with 10 and ending with 15. The data was then printed out per trial in a file corresponding to its size for each array. A summary of the results after statistical analysis are shown in Tables 1 and 2 in the Appendices.

The realtime of v2 is approximately four times that of v1, indicating clear effectiveness of the looping through rows first and then columns. As the array size gets larger, so do the average realtimes, and there are more discrepancies between the realtimes as noted with the increasing standard deviation. Figure 4 compares the percentiles of realtime for the arrays based on size. The relationship between array size and runtime is akin to an exponential function. It is worthy to note that for small array sizes, there isn't much noticeable difference in realtime, however for larger array sizes, there is an apparent difference in realtime for v1 and v2.

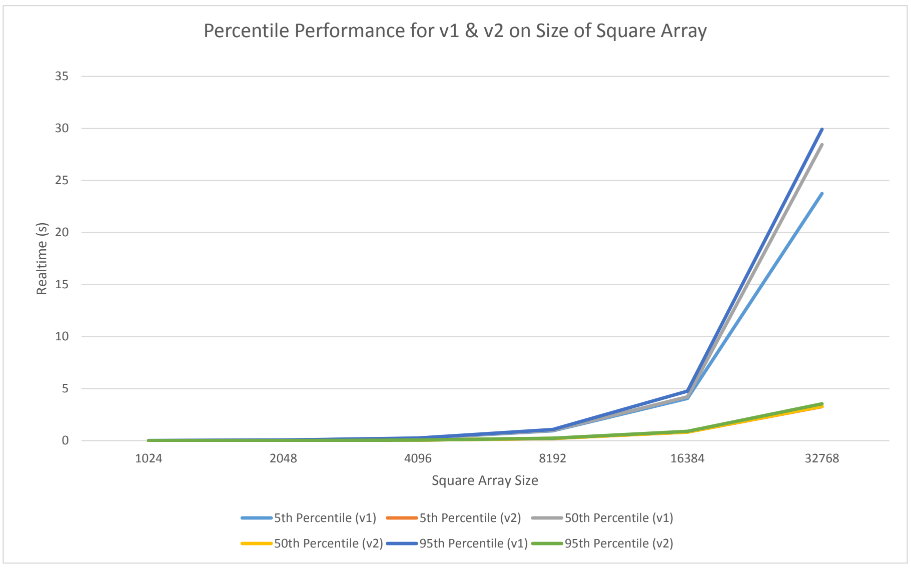


Figure 4: The graph of the 5th, 50th, and 95th Percentile realtimes of each square array in order of increasing powers of 2 based on twenty trials each.

3.2 Non-square Arrays

For non-square arrays, the dimensions of the rows and the columns of the arrays are not equal in value. The width of the array was taken to be half of SIZE, in order to make the dimensions not match and agree with memory allocation. Based on the benchmark bashscript, we performed a total of twenty trials per array size in powers of 2 starting with 10 and ending with 15. The data was then printed out per trial in a file corresponding to its size for each array. A summary of the results after statistical analysis are shown in Tables 3 and 4 in the Appendices.

The realtime of v2 is also approximately four times that of v1 in this case, indicating clear effectiveness of the looping through rows first and then columns. As the array size gets larger, so do the average realtimes, and there are more discrepancies between the realtimes as noted with the increasing standard deviation. Figure 5 compares the percentiles of realtime for the arrays based on size. The relationship between array size and runtime is akin to an exponential function. It is worthy to note that for small array sizes, there isn't much noticeable difference in realtime, however for larger array sizes, there is an apparent difference in realtime for v1 and v2, much like that of square arrays.

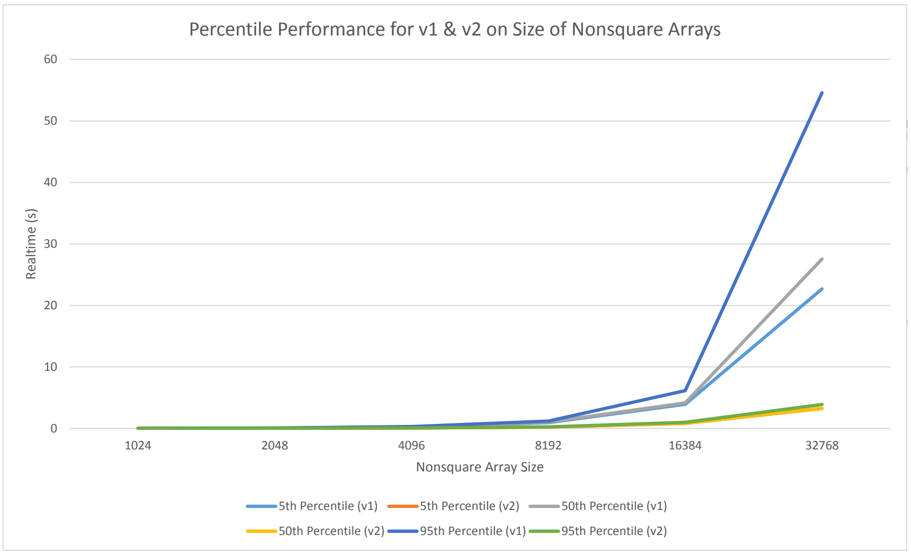


Figure 5: The graph of the 5th, 50th, and 95th Percentile realtimes of each non-square array in order of increasing powers of 2 based on twenty trials each.

4 Conclusion

We have presented an analysis of the two C programs in relation to MIPS while performing an experimental simulation on runtimes for arrays initializing columns first and then rows and vice versa. The reasons for the difference in performance allude to the way memory is structured in computer architecture with the rows stored sequentially first before columns so that when iterating through rows first, all the data is retrieved as opposed to iterating through columns, which would load rows' worth of data per each access request. Further we have demonstrated the MIPS assembly code to explain this phenomena.

References

[1] D. A. Patterson and J.L. Hennessy, *Computer Organization and Design.* Morgan Kaufman 2014.

[2] Ivan Chowdhury and Daniel Nakhimovich, for help with how to do the benchmark, and Andrey Akhmetov for help with how to convert C code into assembly code

A Appendix

Array	v1 (1024)	v2 (1024)	v1(2048)	v2 (2048)	v1 (4096)	v2 (4096)	v1 (8192)	v2 (8192)
Mean Realtime (s)	0.004	0	0.042	0.011	0.236	0.052	0.987	0.216
Standard Deviation	0.004	0	0.004	0.002	0.008	0.004	0.039	0.01
5th Percentile	0	0	0.04	0.01	0.22	0.05	0.94	0.21
50th Percentile	0	0	0.04	0.01	0.23	0.05	0.98	0.21
95th Percentile	0.01	0	0.05	0.01	0.25	0.06	1.07	0.23

Table 1: Statistic measures of the data obtained based on twenty trials for the realtimes in seconds of square arrays of increasing powers of 2 from 10 to 13. We can see that as the array gets larger, so do the realtimes and precision of performance.

Array	v1 (16384)	v2 (16384)	v1(32768)	v2 (32768)
Mean Realtime (s)	4.293	0.840	27.55	3.357
Standard Deviation	0.286	0.027	2.301	0.107
5th Percentile	4.05	0.82	23.75	3.27
50th Percentile	4.22	0.82	28.45	3.3
95th Percentile	4.76	0.89	29.92	3.54

Table 2: Statistic measures of the data obtained based on twenty trials for the realtimes in seconds of square arrays of increasing powers of 2 from 14 to 15. We can see that as the array gets larger, so do the realtimes and precision of performance.

Array	v1 (1024)	v2 (1024)	v1(2048)	v2 (2048)	v1 (4096)	v2 (4096)	v1 (8192)	v2 (8192)
Mean Realtime (s)	0.006	0	0.043	0.011	0.248	0.054	1.031	0.224
Standard Deviation	0.005	0	0.005	0.002	0.020	0.005	0.104	0.023
5th Percentile	0	0	0.04	0.01	0.22	0.05	0.92	0.20
50th Percentile	0.01	0	0.04	0.01	0.24	0.05	1	0.21
95th Percentile	0.01	0	0.05	0.01	0.28	0.06	1.18	0.25

Table 3: Statistic measures of the data obtained based on twenty trials for the realtimes in seconds of non-square arrays of increasing powers of 2 from 10 to 13. We can see that as the array gets larger, so do the realtimes and precision of performance.

Array	v1 (16384)	v2 (16384)	v1(32768)	v2 (32768)
Mean Realtime (s)	4.758	0.873	33.503	3.499
Standard Deviation	0.936	0.079	11.388	0.279
5th Percentile	3.91	0.80	22.68	3.24
50th Percentile	4.16	0.82	27.55	3.28
95th Percentile	6.15	0.98	54.59	3.89

Table 4: Statistic measures of the data obtained based on twenty trials for the realtimes in seconds of non-square arrays of increasing powers of 2 from 14 to 15. We can see that as the array gets larger, so do the realtimes and precision of performance.