UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

VISUALIZATION OF THE MEMORY ACCESS PATTERN OF APPLICATIONS

Diploma Thesis

Kleio Gkoutzomitrou

Supervisor: Christos D. Antonopoulos

Volos 2021

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

VISUALIZATION OF THE MEMORY ACCESS PATTERN OF APPLICATIONS

Diploma Thesis

Kleio Gkoutzomitrou

Supervisor: Christos D. Antonopoulos

Volos 2021

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΟΠΤΙΚΟΠΟΙΗΣΗ ΜΟΤΙΒΩΝ ΠΡΟΣΠΕΛΑΣΗΣ ΕΦΑΡΜΟΓΩΝ ΣΤΗΝ ΜΝΗΜΗ

Διπλωματική Εργασία

Κλειώ Γκουτζομήτρου

Επιβλέπων: Χρήστος Αντωνόπουλος

Βόλος 2021

Approved by the Examination Committee:

Supervisor	**Christos D. Antonopoulos**
	Assistant Professor, Department of Electrical and Computer Engineering, University of Thessaly
Member	**Panagiota Tsompanopoulou**
	Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly
Member	**Aspasia Daskalopoulou**
	Assistant Professor, Department of Electrical and Computer Engineering, University of Thessaly

Date of approval: 16-7-2021

Acknowledgements

I would like to thank my supervisor Christos D. Antonopoulos for his help and guidance throughout this work. I would also like to thank my family for their constant support throughout these five years of study. Last but not least, I want to thank my friends for being always by my side.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Abstract

As processor power continues to exceed memory speed, paired with developers's concern for memory characteristics and hierarchy, tools to analyze memory access behavior will become increasingly important for optimizing data-intensive program execution. Memory access patterns, or how a system or a program reads and writes to memory, vary in their amount of locality of reference and have a significant impact on performance. Identifying access patterns, and then using that information to structure I/O operations and pick appropriate data structures to make the code cache-friendly, can considerably speed up the program's execution.

The goal of this thesis is to create and construct a tool for evaluating and displaying application memory activity and access patterns. The tool we provide offers a comprehensive visual representation of memory access behavior, while visualizing inter access distance, total memory accesses and memory accesses while the program is being executed. This type of visualization makes it simple for the developer to see why performance issues arise and aids in the restructuring of data and code.

Περίληψη

Καθώς η ισχύς του επεξεργαστή συνεχίζει να υπερβαίνει την ταχύτητα της μνήμης, σε συνδυασμό με την ανησυχία των προγραμματιστών για τα χαρακτηριστικά και την ιεραρχία της μνήμης, τα εργαλεία για την ανάλυση της συμπεριφοράς πρόσβασης στη μνήμη θα γίνουν ολοένα και πιο σημαντικά για τη βελτιστοποίηση της εκτέλεσης προγραμμάτων μεγάλου όγκου δεδομένων. Τα μοτίβα πρόσβασης στη μνήμη, δηλαδή ο τρόπος με τον οποίο ένα σύστημα ή ένα πρόγραμμα διαβάζει και γράφει στη μνήμη, ποικίλλει ως προς το μέγεθος της περιοχής αναφοράς και έχει σημαντικό αντίκτυπο στην απόδοση. Ο προσδιορισμός των μοτίβων πρόσβασης και, στη συνέχεια, η χρήση αυτών των πληροφοριών για τη δομή των λειτουργιών I/O και την επιλογή κατάλληλων δομών δεδομένων για να γίνει ο κώδικας φιλικός προς την μνήμη, μπορεί να επιταχύνει σημαντικά την εκτέλεση του προγράμματος.

Ο στόχος αυτής της διατριβής είναι να δημιουργήσει και να κατασκευάσει ένα εργαλείο για την αξιολόγηση και την οπτικοποίηση της δραστηριότητας μνήμης εφαρμογών και των προτύπων πρόσβασης στη μνήμη. Το εργαλείο που παρέχουμε προσφέρει μια ολοκληρωμένη οπτική αναπαράσταση της συμπεριφοράς πρόσβασης στη μνήμη, ενώ απεικονίζει την απόσταση μεταξύ διαδοχικών προσβάσεων στην ίδια διεύθυνση μνήμης, τις συνολικές προσβάσεις στη μνήμη και τις προσβάσεις στη μνήμη κατά την εκτέλεση του προγράμματος ανά διεύθυνση. Αυτός ο τύπος οπτικοποίησης διευκολύνει τον προγραμματιστή να δει γιατί προκύπτουν ζητήματα απόδοσης και βοηθά στην αναδιάρθρωση δεδομένων και κώδικα.

Table of contents

List of figures

List of tables

Abbreviations

GUI	Graphical User Interface
CPU	Central Processing Unit
DBI	Dynamic Binary Instrumentation
DBA	Dynamic Binary Analysis

Chapter 1

Introduction

1.1 Problem Description

In embedded and high-performance computer systems, memory is a key performance barrier. The falling cost of memory and storage, as well as its rising capacity, has resulted in a significant increase in the amount of data processed by programs. Even in data-intensive applications, this fast expansion in main memory has effectively phased out disk I/O, causing memory speed to become the primary focus of optimization. Any type of data processing, gathering, analysis, and even virtualization service are now all competitors for memory use.

Despite the fact that real-world applications contain numerous memory references to a diverse set of data structures, a significant portion of all memory accesses in the application is generated by a few memory instructions with observable, well-known access patterns. This opens the door to memory customization, which can be tailored to the needs of these access patterns. One good way to solve this memory problem is to recognize memory access patterns [3].

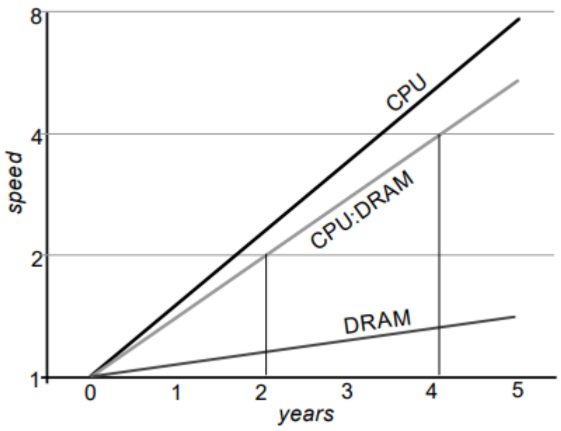


Figure 1.1: Increasing gap between processor and memory. [1]

The purpose of this Thesis is the implementation of a tool that recognizes and visualizes the memory access patterns. This tool will be helpful for programmers to handle the memory more efficient. The tool uses Gleipnir, a Valgrints tool that generates memory traces. Also, it creates three different kinds of graphs.

1.2 Organization of Thesis

This Thesis is separated in five parts:

- In the first part, Chapter 1, we present the problem and give information about the thesis structure and content. Also, we make reference to related work.
- In the second part, Chapter 2, we give information about the background. Memory locality and memory access patterns are being analyzed here. We also give information about Gleipnir tool and the python library Plotly.
- In the third part, Chapter 3, we analyze every step of the implementation of our tool. We begin with memory tracing, pre-proccessing of the data and finally, visualization. We also present the user interface we created and give details about it’s functionality.
- In the fourth part, Chapter 4, we demonstrate examples of our tool usage and we evaluate the results.
- In the fifth part, Chapter 5, we make a conclusion of the Thesis and express thoughts for future work.

1.3 Related Work

Valgrind is a framework for creating dynamic analysis tools. Valgrind's tools can automatically discover a wide range of memory management and threading problems, as well as profile your programs in great detail. Valgrind may also be used to create new tools.

Valgrind is a GNU-licensed open-source DBI framework that is being developed and maintained by many people across the world. Datagrind, mmtrace, memhist, and memview are examples of notable tools developed around this framework that specialize on memory access profiling, comparable to Gleipnir. They are all created by independent developers with the purpose of collecting a program's memory trail and giving some basic statistics for accesses.

An attempt, similar to the one we will analyze in this thesis, was made in the thesis of Christos Ntogkas [4]. In his thesis, he used trace files created using Gleipnir and Cachegrind. In his Thesis, he presents three types of visualizations. The metrics he presents in the visualizations are:

- reuse distance
- average reuse distance
- reuse distances standard deviation

Chapter 2

Background

2.1 Locality of reference

Locality of reference, often known as the principal of locality, is one of the most essential aspects of memory access patterns. It refers to a processor's tendency to repeatedly access the same set of memory locations over time. There are two fundamental kinds of locality of reference, temporal locality and spatial locality [5].

Temporal locality refers to a program's tendency to reuse data elements many times throughout a short period of time during execution. Loops, for example, repeatedly fetch the same instructions. Calling and returning from functions, for instance, causes stack memory to be accessed frequently. This is the underlying idea of caching, and it provides a clear path to a suitable data-management heuristic. The only actual constraint to utilizing this type of locality is cache storage capacity [6].

Spacial locality refers to the fact that if a certain data element is accessed at a given moment, it's likely that nearby memory locations will be referenced soon after. Arrays of data that are accessed sequentially are an excellent illustration of this sort of locality.

These two types of locality are represented in the Figure 2.1. This example is a good example of locality.

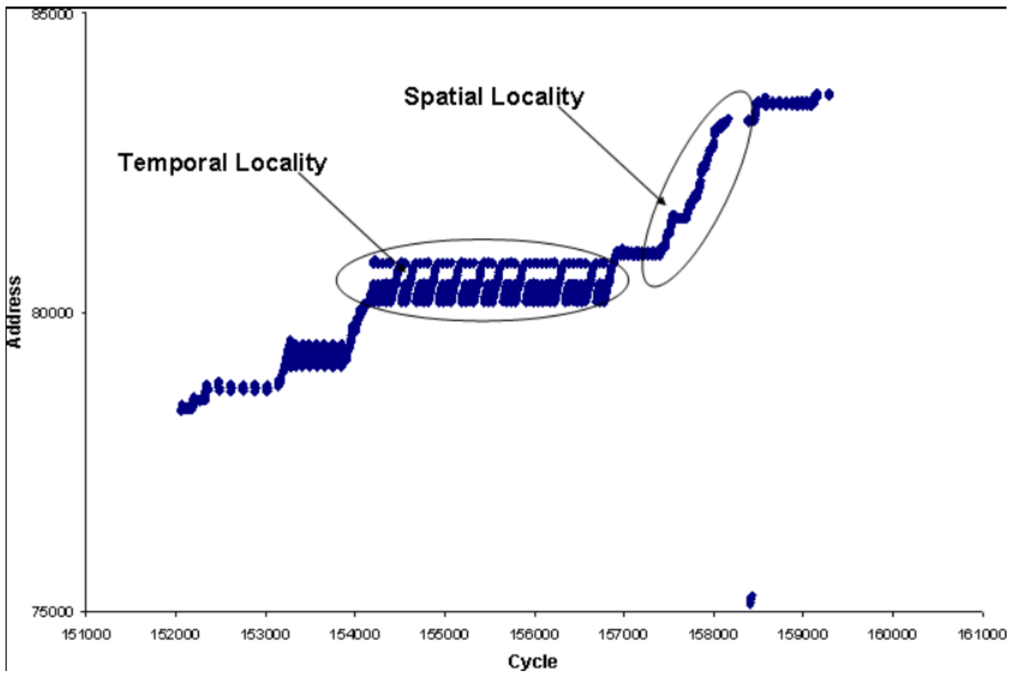


Figure 2.1: Example of good locality. [2]

2.2 Memory Access Pattern

The major performance barrier nowadays, thanks to improvements in circuit design and new lithography technologies, is memory operations, particularly secondary memory, often known as the ”Von Neumann bottleneck” [7]. Although ’random access’ is commonly used to characterize computer memory, software traversal will nonetheless reveal patterns that might be exploited for efficiency. While the structure and functioning of different software may appear to be identical, the memory access patterns can vary greatly. The number of memory access patterns is practically limitless. However, there are a few memory access patterns that occur frequently.

- Sequential

  Sequential access pattern is the simplest one and refers to reads and writes on increment or decremented straightforward addresses.

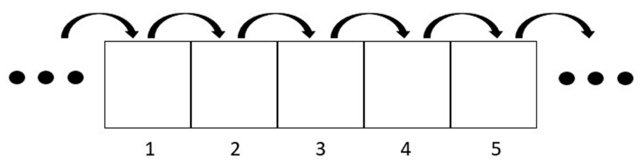


Figure 2.2: Sequential Memory Access Pattern

- Strided

  Strided access pattern with stride K indicates accessing every Kth memory element. If stride equal to 1 strided access pattern is equivalent to sequential access pattern [4].

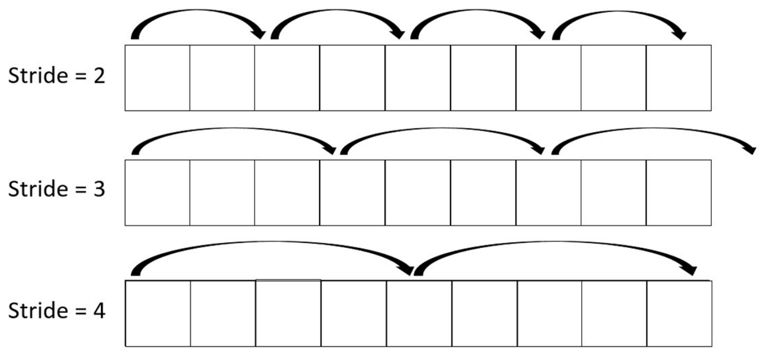


Figure 2.3: Strided Memory Access Pattern

- Linear

  A linear access pattern is similar to ”strided,” in which a memory address can be calculated using a linear combination of indexes. In systems that allow to compute kernels, a linear access pattern for writes (with any access pattern for non-overlapping reads) may guarantee that an algorithm may be parallelized [8].

- Random

  Cache and memory performance are harmed by random memory access patterns. Randomness is not characterized in this context as fully arbitrary addressing, but rather accesses that are not consecutive, do not access the same data or data that has been recently accessed, and do not follow a consistent pattern that may be recognized and exploited by the hardware prefetcher.

- Scatter

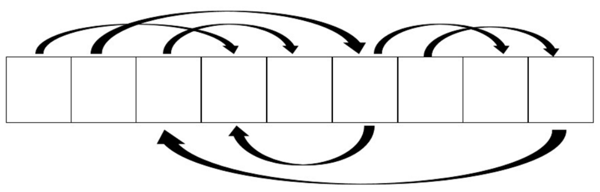


Figure 2.4: Random Memory Access Pattern

The scatter access pattern refers to accesses that occurs in a sequential order when reading, but at random when writing. However, because there is no guarantee that writes to memory are not independent, parallelization may be a difficult task.

2.3 Gleipnir Tool

Valgrind is a dynamic binary instrumentation (DBI) framework [9]. It is made for creating powerful Dynamic Binary Analysis (DBA) tools. Memory debugging, memory leak detection, and profiling are all possible with it. Valgrind lies between the program's execution layer and the operating system. When the user executes the program, the Valgrind core passes it to a suite of tools chosen by the user before executing the instructions.

We use Gleipnir [10], a plug-in for Valgrind, to acquire the memory traces. Trace created by Gleipnir, consists load, store, and modify instructions tracked down to source level variables for stack and global data recognizable by debug information parsing. The other types of instructions are considered as unidentified kinds.

Gleipnir adopts the dynamic instrumentation approach, which, although adding considerable runtime overhead and slowing execution by tens or hundreds of times, but it gives more information and flexibility than the alternatives. So we came to the conclusion that Gleipnir was the right choice for our tool.

2.4 Plotly

Plotly is an interactive, open-source, Python plotting library [11]. It supports over 40 different chart types for statistical, financial, geographic, scientific, and 3-dimensional applications. Plotly allows Python programmers to generate stunning interactive web-based visualizations that may be viewed in Jupyter notebooks, saved as independent HTML files, or delivered as part of pure Python-built web apps using Dash. The reason why we choose Plotly is that it is really simple library to use, but in the same time it offers the option of every visualization you may need to create.

2.5 Visualization Metrics

- **Inter Access Distance**

  For this visualization, we need to introduce the terms inter access distance and standard deviation. Measuring inter access distance between two data visits, means counting the number of distinct data accesses between them. The standard deviation is a measurement of a collection of values' variance or dispersion [12].

  This type of visualization is useful to understand the way that each address is being accessed.

- **Total Accesses per Address**

  For this type of visualization, we counted every access that occurred for each address. This information is crucial to determine which addresses are most often used. Having this information, developers can use variables in such a way that the code can be more efficient.

- **Accesses during Execution**

  Visualizing the accesses during the execution of the program, gives us the opportunity to observe when the most accesses occur. Also, memory locality is noticeable in this kind of graph.

Chapter 3

Implementation

In terms of optimization, memory performance has taken center stage. As a result, more tools for profiling and evaluating applications are being developed. These tools frequently use the approach of observing the code of an application as it is being executed and collecting information about its inner procedures. There are three ways to do this for memory behavior: first, inserting code capable of collecting the required information at compile time, known as static instrumentation, second, inserting code after the compilation that examines the instructions and collects the required data, known as dynamic instrumentation, and third, simulating the memory hierarchy [4].

Figure 3.1 presents the creation stages of our visualization tool. These stages are going to be analyzed in this chapter.

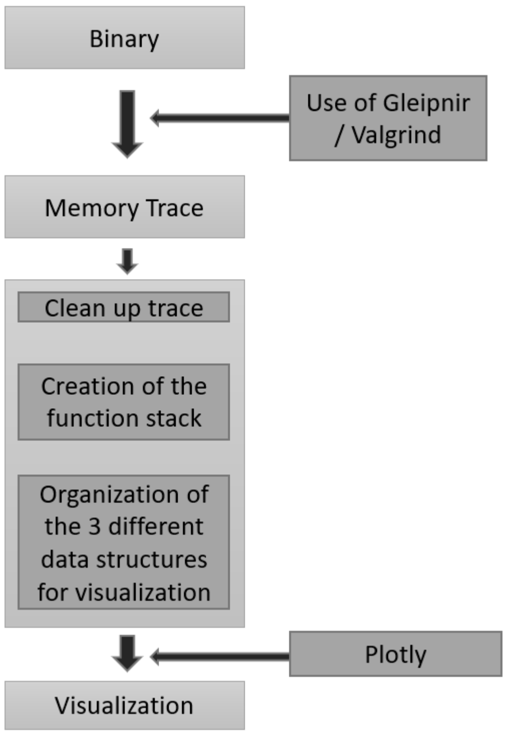
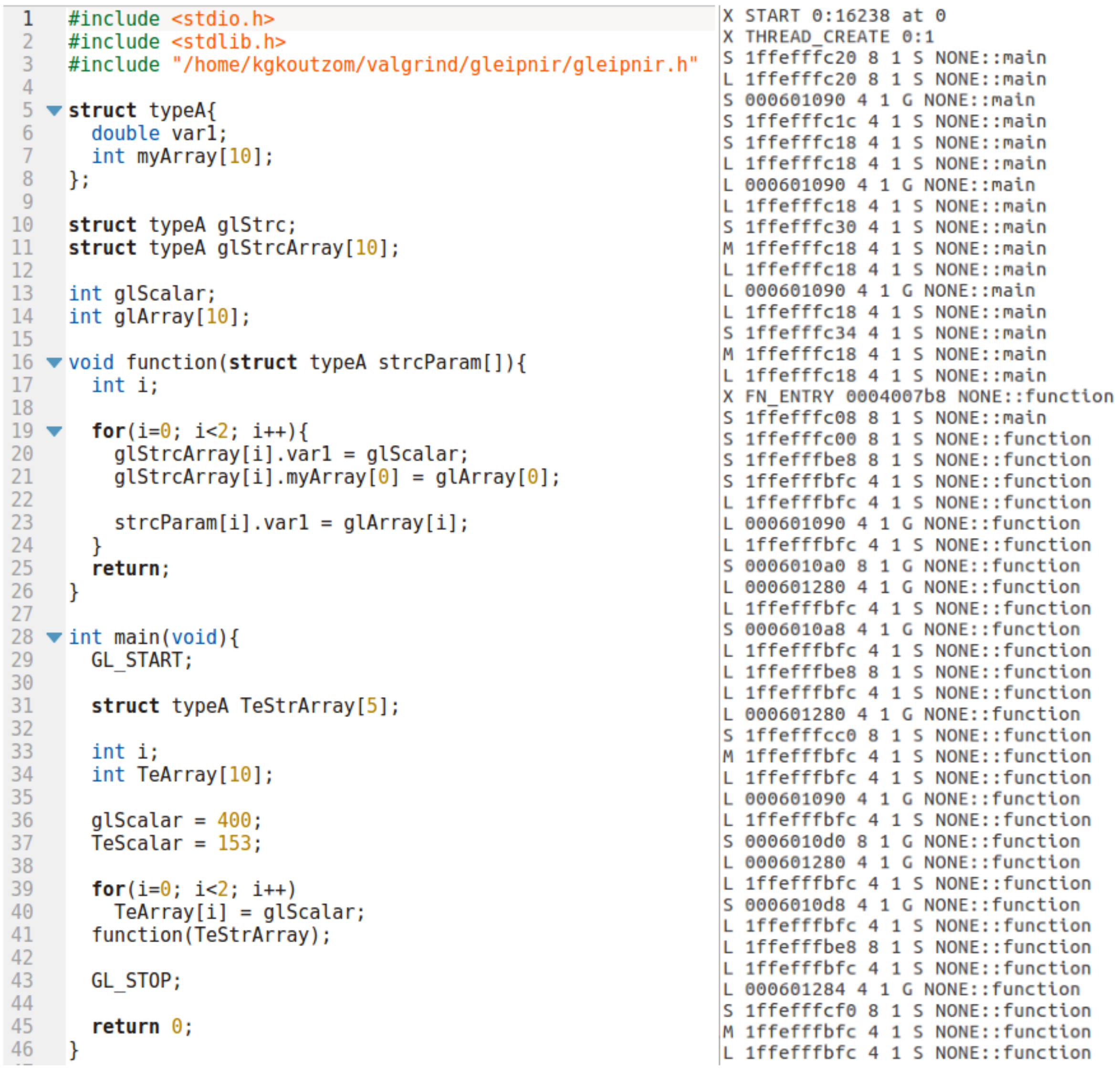

Binary
Use of Gleipnir / Valgrind
Memory Trace
Clean up trace
Creation of the function stack
Organization of the 3 different data structures for visualization
Plotly
Visualization

Figure 3.1: Stages of Implementation

3.1 Memory trace

Gleipnir, a plug-in built for the famous binary instrumentation tool Valgrind, was used to obtain the memory traces. Valgrind is a dynamic binary instrumentation (DBI) framework with a distinct design space. It's made for creating heavyweight.Dynamic Binary Analysis (DBA) tools. Memory debugging, memory leak detection, and profiling are all possible with it. [13].



```c
#include <stdio.h>
#include <stdlib.h>
#include "/home/kgkoutzom/valgrind/gleipnir/gleipnir.h"

struct typeA{
  double var1;
  int myArray[10];
};

struct typeA glStrc;
struct typeA glStrcArray[10];

int glScalar;
int glArray[10];

void function(struct typeA strcParam[]){
  int i;

  for(i=0; i<2; i++){
    glStrcArray[i].var1 = glScalar;
    glStrcArray[i].myArray[0] = glArray[0];

    strcParam[i].var1 = glArray[i];
  }
  return;
}

int main(void){
  GL_START;

  struct typeA TeStrArray[5];

  int i;
  int TeArray[10];

  glScalar = 400;
  TeScalar = 153;

  for(i=0; i<2; i++)
    TeArray[i] = glScalar;
  function(TeStrArray);

  GL_STOP;

  return 0;
}
```

```
X START 0:16238 at 0
X THREAD_CREATE 0:1
S 1ffefffc20 8 1 S NONE::main
L 1ffefffc20 8 1 S NONE::main
S 000601090 4 1 G NONE::main
S 1ffefffc1c 4 1 S NONE::main
S 1ffefffc18 4 1 S NONE::main
L 1ffefffc18 4 1 S NONE::main
L 000601090 4 1 G NONE::main
L 1ffefffc18 4 1 S NONE::main
S 1ffefffc30 4 1 S NONE::main
M 1ffefffc18 4 1 S NONE::main
L 1ffefffc18 4 1 S NONE::main
L 000601090 4 1 G NONE::main
L 1ffefffc18 4 1 S NONE::main
S 1ffefffc34 4 1 S NONE::main
M 1ffefffc18 4 1 S NONE::main
L 1ffefffc18 4 1 S NONE::main
X FN_ENTRY 0004007b8 NONE::function
S 1ffefffc08 8 1 S NONE::main
S 1ffefffc00 8 1 S NONE::function
S 1ffefffbe8 8 1 S NONE::function
S 1ffefffbfc 4 1 S NONE::function
L 1ffefffbfc 4 1 S NONE::function
L 000601090 4 1 G NONE::function
L 1ffefffbfc 4 1 S NONE::function
S 0006010a0 8 1 G NONE::function
L 000601280 4 1 G NONE::function
L 1ffefffbfc 4 1 S NONE::function
S 0006010a8 4 1 G NONE::function
L 1ffefffbfc 4 1 S NONE::function
L 1ffefffbe8 8 1 S NONE::function
L 1ffefffbfc 4 1 S NONE::function
L 000601280 4 1 G NONE::function
S 1ffefffcc0 8 1 S NONE::function
M 1ffefffbfc 4 1 S NONE::function
L 1ffefffbfc 4 1 S NONE::function
L 000601090 4 1 G NONE::function
L 1ffefffbfc 4 1 S NONE::function
S 0006010d0 8 1 G NONE::function
L 000601280 4 1 G NONE::function
L 1ffefffbfc 4 1 S NONE::function
S 0006010d8 4 1 G NONE::function
L 1ffefffbfc 4 1 S NONE::function
L 1ffefffbe8 8 1 S NONE::function
L 1ffefffbfc 4 1 S NONE::function
L 000601284 4 1 G NONE::function
S 1ffefffcf0 8 1 S NONE::function
M 1ffefffbfc 4 1 S NONE::function
L 1ffefffbfc 4 1 S NONE::function
```

Figure 3.2: Source Code and Gleipnir Trace

Figure 3.2 presents the source code of a simple program and the trace we collect from Gleipnir tool. From each line of the trace file, we observe the following data:

```
OperationType Address MemorySize ThreadId Scope Function
```

- OperationType: This element determines the type of access. L(Load), S(Store), M(Modify), X(Special Operation, such as entry of a function).
- Address: This element specifies the address to which it is accessed.
- MemorySize: This element specifies the memory size that being accessed.
- ThreadId: This element specifies the ID of the thread that is responsible for the access.

- Scope: This element specifies the position. G(Global), H(Heap), S(Stack).

- Function: This element specifies the name of the function that is responsible for the access.

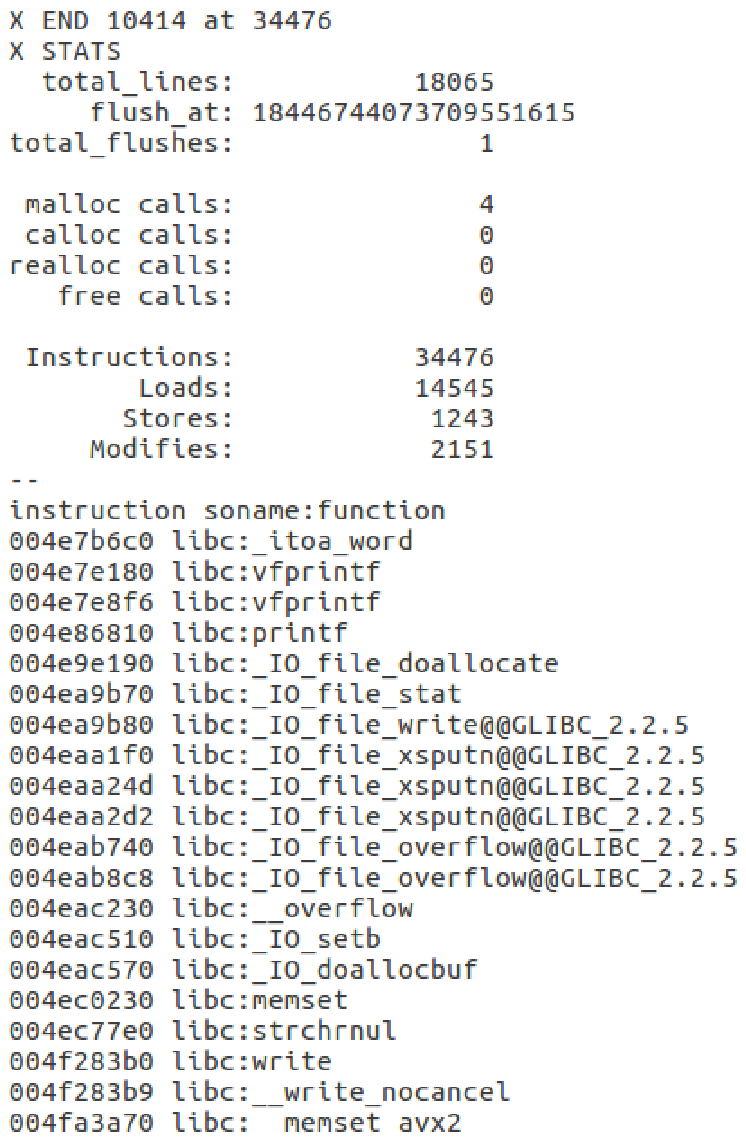
One of the disadvantages of Gleipnir is that it produces very large files relative to the size of the running program. For example, the multiplication of two 150x150 matrix, which is a small example of source code, will produce a trace of 1.5 GB.

3.2 Pre-Proccessing

Gleipnir, as previously indicated, generates huge trace files. Because we don't have limitless memory, we will need to figure out a technique to decrease the trace file's size. The next step is to create the program stack, in order to define the decedents of each function. The final step is to organize the data into appropriate structures for each type of visualization.

3.2.1 Cleaning up the trace file:

In order to reduce the size of the trace file, we removed some lines that were not useful for our visualizations. The lines with operation type X were deleted, except for those in which X indicated entry to a function. We also deleted the last lines of the trace file. These lines indicate the end of the file and give some general information about the trace, which is not required for our visualizations. Although this information is not useful for our visualizations, they provide a general picture of the trace 3.3.



```
X END 10414 at 34476
X STATS
  total_lines:          18065
     flush_at: 18446744073709551615
total_flushes:              1

 malloc calls:              4
 calloc calls:              0
realloc calls:              0
   free calls:              0

 Instructions:          34476
        Loads:          14545
       Stores:           1243
     Modifies:           2151
--
instruction soname:function
004e7b6c0 libc:_itoa_word
004e7e180 libc:vfprintf
004e7e8f6 libc:vfprintf
004e86810 libc:printf
004e9e190 libc:_IO_file_doallocate
004ea9b70 libc:_IO_file_stat
004ea9b80 libc:_IO_file_write@@GLIBC_2.2.5
004eaa1f0 libc:_IO_file_xsputn@@GLIBC_2.2.5
004eaa24d libc:_IO_file_xsputn@@GLIBC_2.2.5
004eaa2d2 libc:_IO_file_xsputn@@GLIBC_2.2.5
004eab740 libc:_IO_file_overflow@@GLIBC_2.2.5
004eab8c8 libc:_IO_file_overflow@@GLIBC_2.2.5
004eac230 libc:__overflow
004eac510 libc:_IO_setb
004eac570 libc:_IO_doallocbuf
004ec0230 libc:memset
004ec77e0 libc:strchrnul
004f283b0 libc:write
004f283b9 libc:__write_nocancel
004fa3a70 libc:  memset avx2
```

Figure 3.3: Part of the Gleipnir trace, last lines.

3.2.2 Creation of the stack:

The creation of a stack of program functions was the next critical step. The creation of the stack was necessary because we need to know the call tree of the functions and the function call tree that lead to the specific accesses. This feature is necessary to visualize accesses from a function and its descendants.

First, we numbered each line of the trace so that it is easy to refer to the stack for each access separately. For each line of the trace with a format like that Figure 3.4 shows, we store function1 in a temporary variable. Because the trace we extracted from Gleipnir only gives us the information that an entry is made to a function, we needed to extract the information that an exit is made from a function. As a result, we examined the trace and determined when a function exit happens, and so we created the functions stack.

```
X FN_ENTRY address libc::function1
```

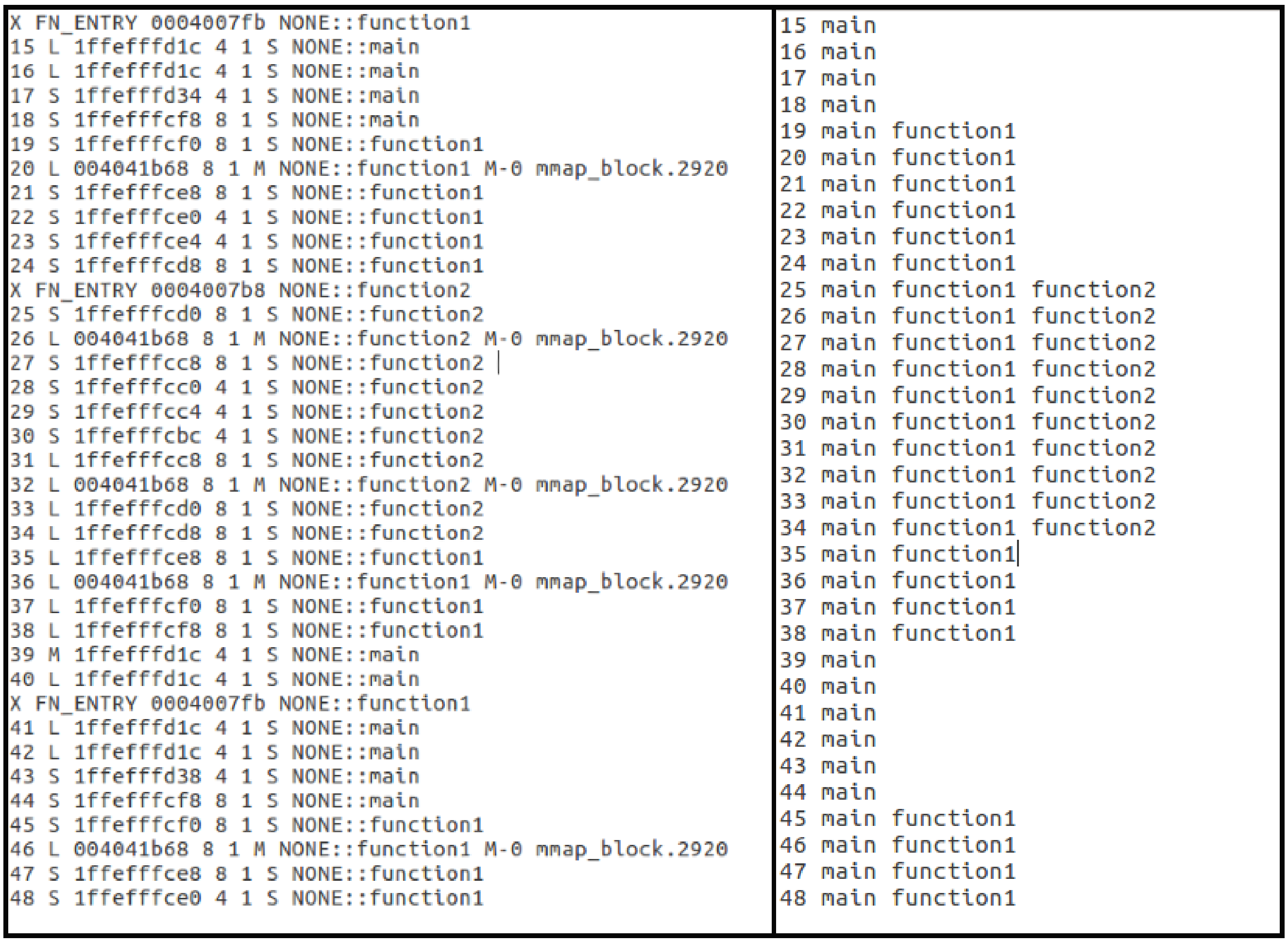
Figure 3.4: Line of the trace that shows enter in function1

Figure 3.5 presents a simple code example. Figure 3.6 presents a piece of the trace and the corresponding stack snapshot. We used this simple example to understand the stages of stack creation, as we will explain later in this chapter.

```
void function2()               void function1()
  array[0]=1;                    array[0]=1;
  array[1]=2;                    array[1]=2;
  i = 1;                         function2();

int main()
  function2();
  for i ← 1 to 5
    arr[i] = i + 1;
    function1();
```

Figure 3.5: C pseudocode for a simple example.



```
X FN_ENTRY 0004007fb NONE::function1
15 L 1ffefffd1c 4 1 S NONE::main
16 L 1ffefffd1c 4 1 S NONE::main
17 S 1ffefffd34 4 1 S NONE::main
18 S 1ffefffcf8 8 1 S NONE::main
19 S 1ffefffcf0 8 1 S NONE::function1
20 L 004041b68 8 1 M NONE::function1 M-0 mmap_block.2920
21 S 1ffefffce8 8 1 S NONE::function1
22 S 1ffefffce0 4 1 S NONE::function1
23 S 1ffefffce4 4 1 S NONE::function1
24 S 1ffefffcd8 8 1 S NONE::function1
X FN_ENTRY 0004007b8 NONE::function2
25 S 1ffefffcd0 8 1 S NONE::function2
26 L 004041b68 8 1 M NONE::function2 M-0 mmap_block.2920
27 S 1ffefffcc8 8 1 S NONE::function2
28 S 1ffefffcc0 4 1 S NONE::function2
29 S 1ffefffcc4 4 1 S NONE::function2
30 S 1ffefffcbc 4 1 S NONE::function2
31 L 1ffefffcc8 8 1 S NONE::function2
32 L 004041b68 8 1 M NONE::function2 M-0 mmap_block.2920
33 L 1ffefffcd0 8 1 S NONE::function2
34 L 1ffefffcd8 8 1 S NONE::function2
35 L 1ffefffce8 8 1 S NONE::function1
36 L 004041b68 8 1 M NONE::function1 M-0 mmap_block.2920
37 L 1ffefffcf0 8 1 S NONE::function1
38 L 1ffefffcf8 8 1 S NONE::function1
39 M 1ffefffd1c 4 1 S NONE::main
40 L 1ffefffd1c 4 1 S NONE::main
X FN_ENTRY 0004007fb NONE::function1
41 L 1ffefffd1c 4 1 S NONE::main
42 L 1ffefffd1c 4 1 S NONE::main
43 S 1ffefffd38 4 1 S NONE::main
44 S 1ffefffcf8 8 1 S NONE::main
45 S 1ffefffcf0 8 1 S NONE::function1
46 L 004041b68 8 1 M NONE::function1 M-0 mmap_block.2920
47 S 1ffefffce8 8 1 S NONE::function1
48 S 1ffefffce0 4 1 S NONE::function1
```

```
15 main
16 main
17 main
18 main
19 main function1
20 main function1
21 main function1
22 main function1
23 main function1
24 main function1
25 main function1 function2
26 main function1 function2
27 main function1 function2
28 main function1 function2
29 main function1 function2
30 main function1 function2
31 main function1 function2
32 main function1 function2
33 main function1 function2
34 main function1 function2
35 main function1
36 main function1
37 main function1
38 main function1
39 main
40 main
41 main
42 main
43 main
44 main
45 main function1
46 main function1
47 main function1
48 main function1
```

Figure 3.6: Example of stack creation.

In the trace file, there are lines doing simple operations, like loads and stores, and also lines with format like in Figure 3.4. Lines with that format suggest the entry of a function.

We started with the stack containing only the main function and then locating the first entry in the function1. For the next 4 lines, our stack continues to consist of the main function. On line 19 the first access is made by function1, and so function1 enters the stack. Then, until we meet the next entry, only function1 causes accesses and our stack consists of main and function1. After entry in function2, the stack for lines 25-34 consists of: main, function1, function2. We notice that function2 was the first function that caused access after its entry, when it entered the stack directly.

At line 35, access is triggered by function1. At that time our stack is: main, function1, function2. In order to be able to access function1, means that function2 has exited. So our stack now consists of main and function1, up to line 38. Then main is the function that causes access. Similarly, the stack now comprises just of the main, for lines 40 and up. Then we come across an entry from function1. However, function1 causes access four accesses later, on line 45, and enters the stack.

3.2.3 Organization of the data:

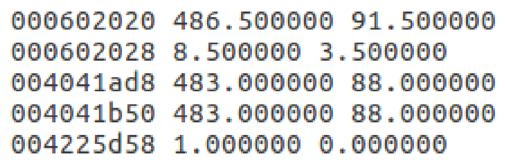
We have filtered out the irrelevant information and arranged the rest into helpful structures. We can now analyze these structures to get useful information about the access pattern of the program. For each visualization, we need a different type of data structure. We will focus on the following type of visualization to give us with a thorough insight into the features of the program's memory access pattern, based on the study we intend to perform:

- **Inter Access Distance**

  For this type of visualization, we used the average value of the inter access distance, and we also calculated the standard deviation for each address. The data structure we created for this visualization, is presented in Figure 3.7 and is of the form:

  ```
  Address AverageAccessDistance StandardDeviation
  ```

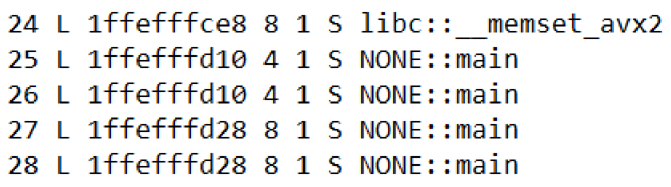
  and consists of a line for every different memory address.



```
000602020 486.500000 91.500000
000602028 8.500000 3.500000
004041ad8 483.000000 88.000000
004041b50 483.000000 88.000000
004225d58 1.000000 0.000000
```

Figure 3.7: Part of data structure for inter access distance.

To create this data structure, we used the Gleipnir trace, after we had numbered each of its lines. Then, since we knew the stack in each access, we kept only the accesses that interested us. Because every line of the trace was numbered, we knew when each access was made (stage1). After that step, the trace file has the type shown it Figure 3.8 (stage2). We only need the first and the third column of each line, so we deleted other information (stage3) and sorted the file according to the third column, which included the memory addresses (stage4). After the shorting, the file contained the number that shows the sequence that the accesses happened for every access, in consecutive lines. Finally, we calculated the inter access distance and the standard deviation by reading the file line by line, and calculating these values for each address (stage5).



```
24 L 1ffefffce8 8 1 S libc::__memset_avx2
25 L 1ffefffd10 4 1 S NONE::main
26 L 1ffefffd10 4 1 S NONE::main
27 L 1ffefffd28 8 1 S NONE::main
28 L 1ffefffd28 8 1 S NONE::main
```

Figure 3.8: Data before sorting and deleting unnecessary information.

- **Total Accesses per Address** For this type of visualization, we followed the same steps as in inter access visualization, until the trace file became like in Figure 3.8, stage2. Then we counted every access that occurred for each address and created a data structure of the type:

  ```
  Address NumberOfAccesses
  ```

- **Timeline of accesses** As we said before, we numbered each line of the trace file. These numbers show the sequence that accesses happen. For this visualization, we followed the same steps as in inter access visualization, until stage4.

3.3 Visualization

The most interesting information our tool provides are the graphs it creates. In this regard, a new graphical user interface (GUI) was designed to make it easier for the user to manage the findings.

Tkinter, an interface to the Tk GUI toolkit [14], is one of the most popular graphical user interface (GUI) widget set for Python. Because of its widespread use, it is included in the Python installation and requires no additional configuration. It is simple to use, supports many windows and widgets, and makes importing and managing graphs an easy task [4].

3.3.1 User Interface:

Now that we have collected all the need data for the graphs and have decided which tool to use for the creation of the interface, it is time to present the GUI. Our purpose was to create a user-friendly interface. Figure 3.9 depicts a view of the completed graphical user interface.

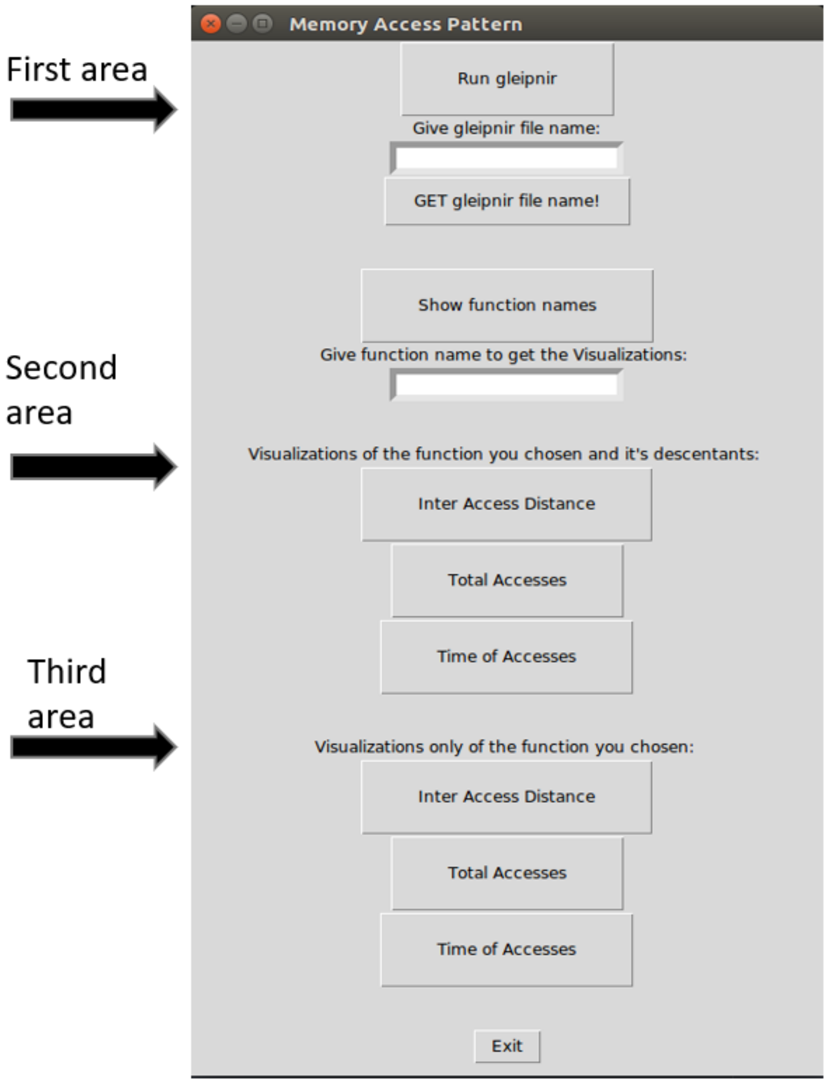


Figure 3.9: Graphic User Interface (GUI)

The interface is separated in three areas.

In the first area, the user can run Gleipnir or, if this procedure has already accomplished, can insert the Gleipnir file name. This may be the most time-consuming part of using GUI, as Gleipnir creates very large files. Also, in this area, pressing the button ”Show function names”, a new window open, and it shows all the function names of the program. Thanks to this feature, the user can choose if he wants to implement graphs for the accesses of a specific function.

In the second and third area, there are three buttons in each. These buttons implement and display the graphs for each of the 3 types of visualizations we have implemented. The reason why there are two areas with the same apparent buttons is because in the second area the user chooses to implement visualization for the function he has inserted in the first area, and its descendants, while in the third area the visualization concerns only the function the user has inserted and not the function's descendants.

Finally, at the bottom of the interface, there is an exit button to close the interface.

3.3.2 Plot Selection:

Plotly, which is a Python plotting library [11], will be used to create the plots stated before.

For the inter access distance visualization, we used an error bar plot. This plot shows the average inter access distance and the standard deviation. Error bars are graphical representations of the variability of data and they give a general idea of how precise a measurement is. In our case, it indicates whether successive accesses to an address are the same distance apart.

A line plot with markers, used for total accesses visualization. We used this type of plot because want to observe the rate of change of accesses amount between addresses. A scatter plot used to visualize accesses during execution of the program. Scatter plot used to show relationships between accesses and time. A scatter plot shows not only the values of individual memory accesses, but also the patterns of memory accesses.

Our choices aimed to simple graphs, so that the user can easily extract information from them.

The tool export the figures to HTML files.

Chapter 4

Evaluation

In this chapter, we demonstrate the capabilities of the tool we created. Matrix multiplication of two 10x10 matrices is the source code utilized. The results are incredibly easy to evaluate due to the table's modest size. For programs that have many memory accesses our tool provide the option of zoom in, to make the graph more clear.

4.1 Plots with descendants and without

In this example, we used the accesses made from function main. In Figure 4.1, Figure 4.2 and Figure 4.3 we observe the plots our tool creates for the main function, and its descendants. In Figures, Figure 4.4, Figure 4.5 and Figure 4.6 we observe the plots that were created only for the accesses caused by main.

By observing these Figures, we can come to some conclusions:

1. The inter access distance and the standard deviation is the same for the most addresses when we visualize accesses from a function and its descendants, than when we visualize only the function's accesses. This is logical, some data structures are used only from one function. For this reason, addresses are less when we visualize inter access distance only for accesses caused by one function.

2. Total accesses follow the same pattern in both of the two visualizations. In Figure 4.2 there are more fluctuations than in Figure 4.5 because the total number of accesses is bigger as well. In this example, function main never make access in some addresses that its descendants make accesses.

3. Accesses timeline of the program follow a random pattern until the 1000th access. After that point, the pattern they follow is sequential for most of the accesses. This observation applies to both visualizations in Figure 4.3 and in Figure 4.6.

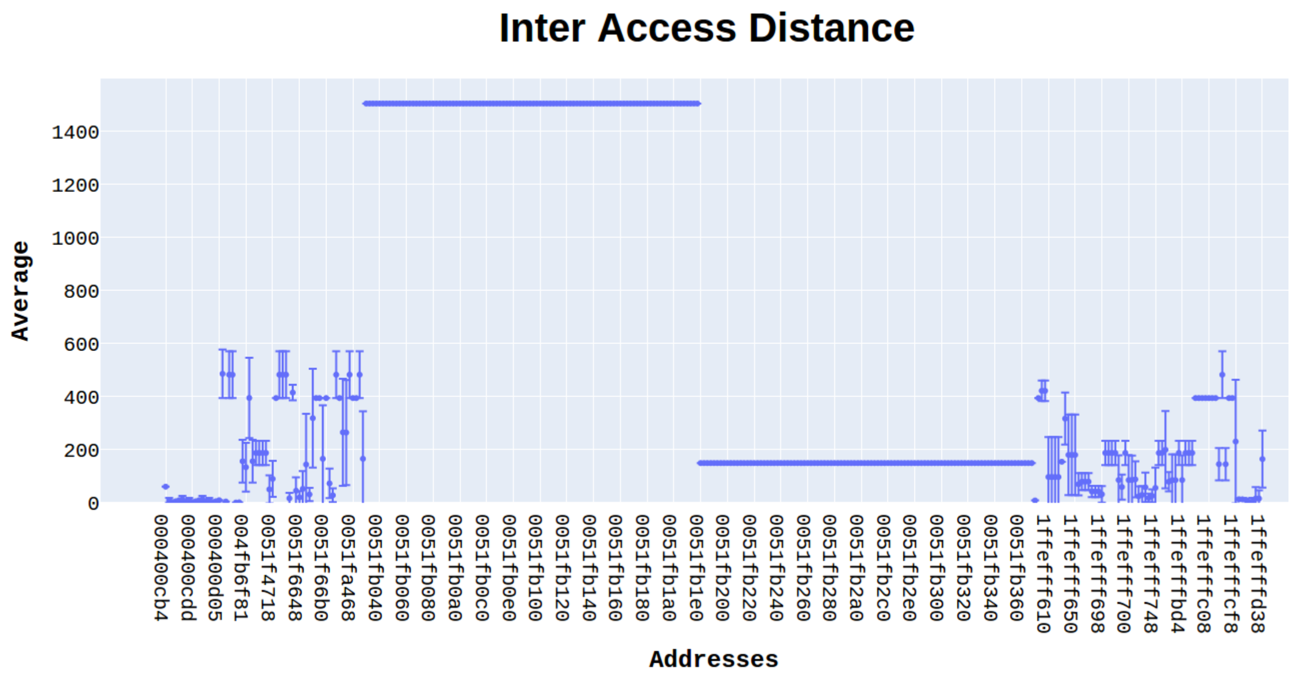


Figure 4.1: Inter access distance for main and its descendants.

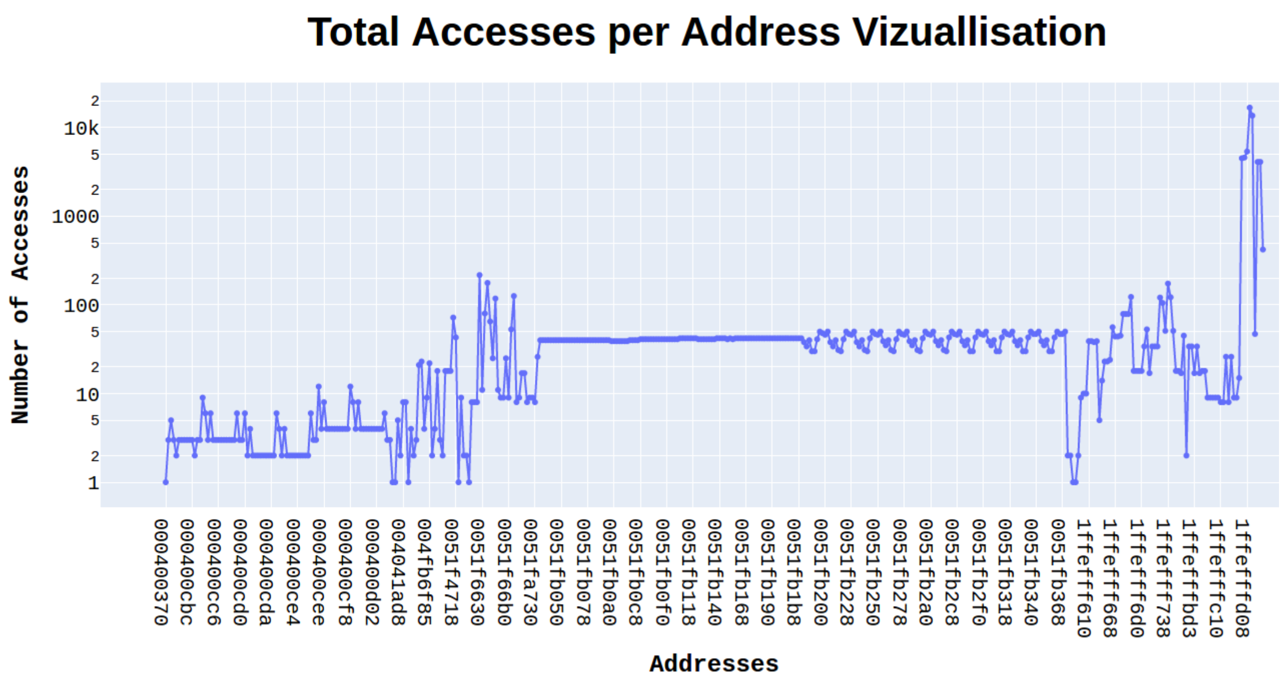


Figure 4.2: Total accesses for main and its descendants.

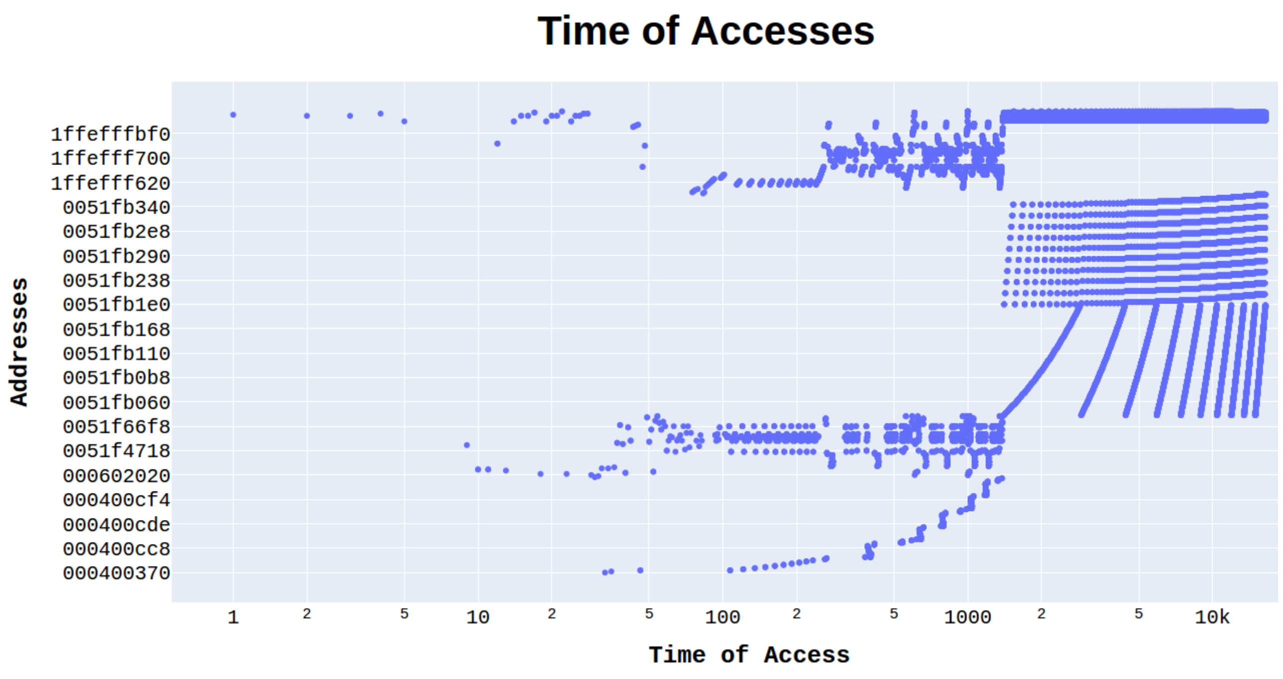


Figure 4.3: Accesses timeline for main and its descendants.

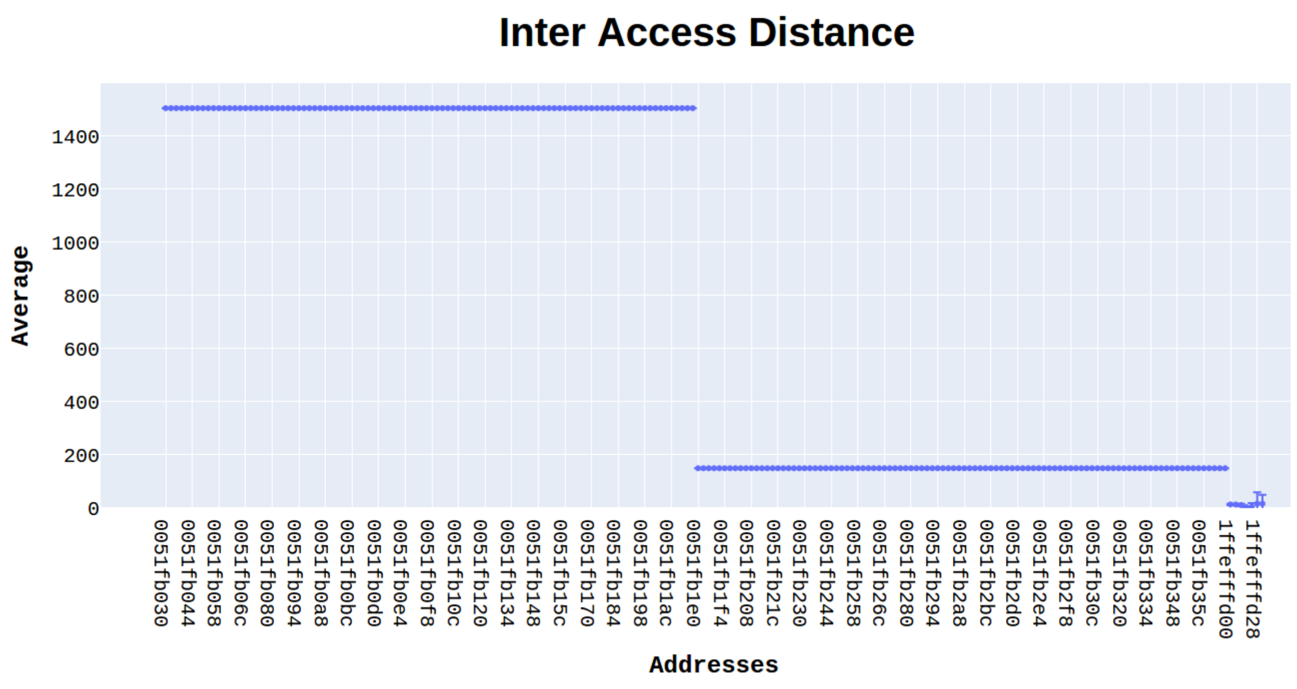


Figure 4.4: Inter access distance for main.

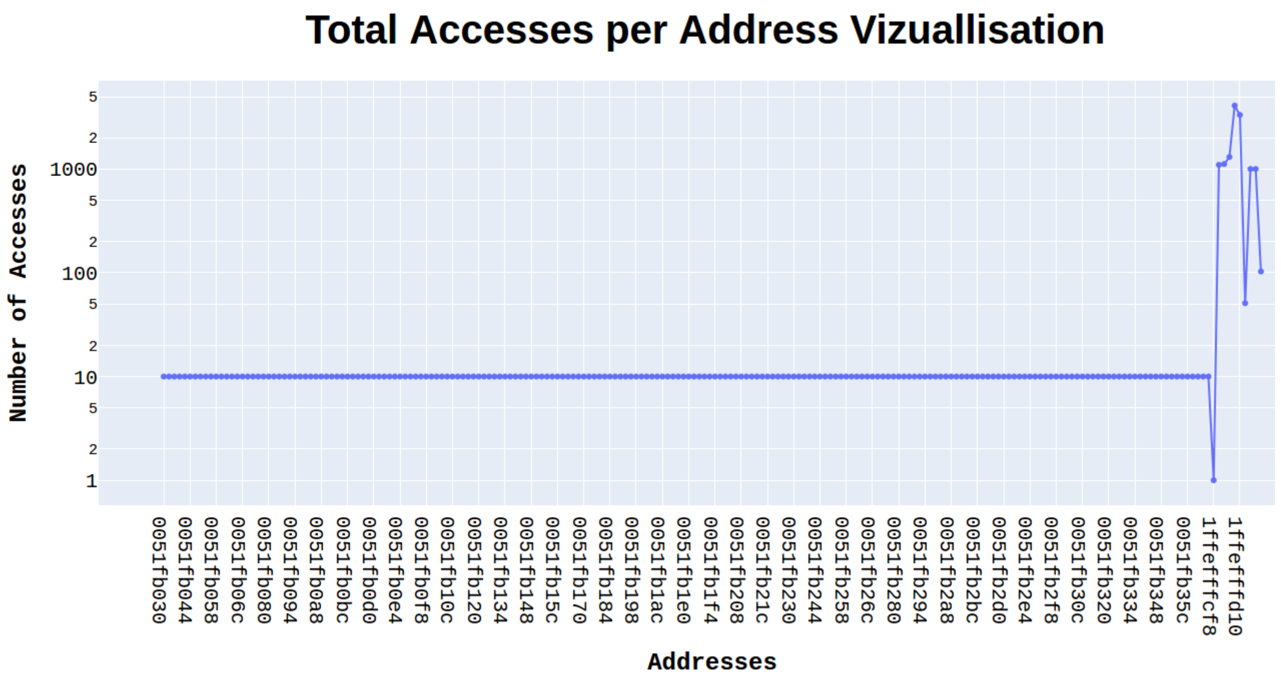


Figure 4.5: Total accesses for main.

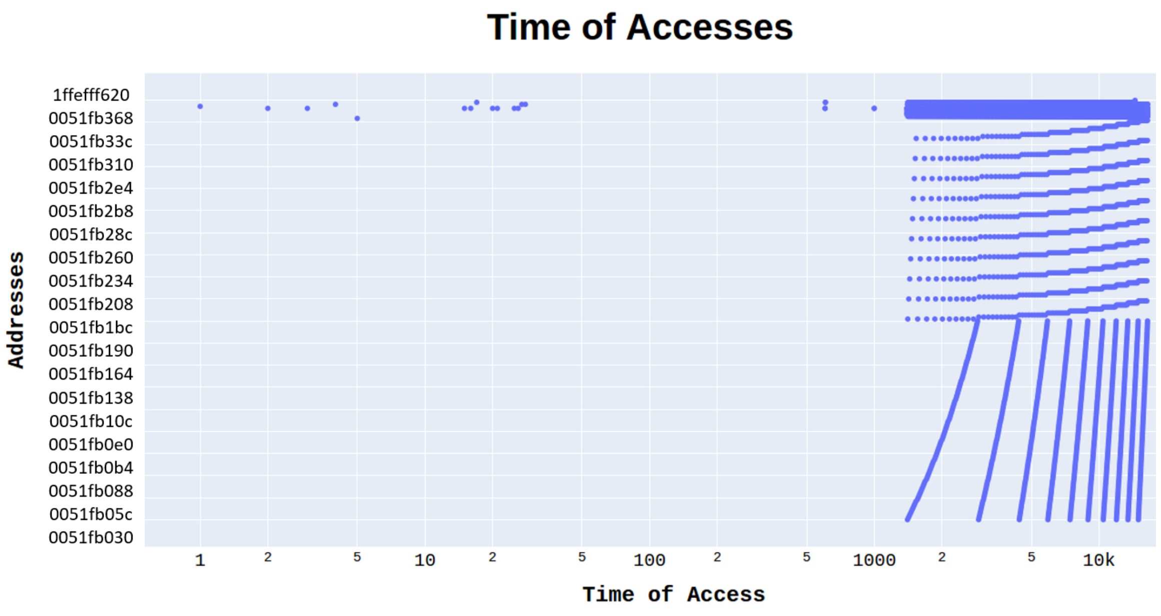


Figure 4.6: Accesses timeline for main.

4.2 Comparison between two types of matrix multiplication

In this section, we compare the memory access patterns between matrix multiplication of 150x150 matrices and blocked matrix multiplication of 150x150 matrices, with block size equal to 10.

4.2.1 Inter Access Distance

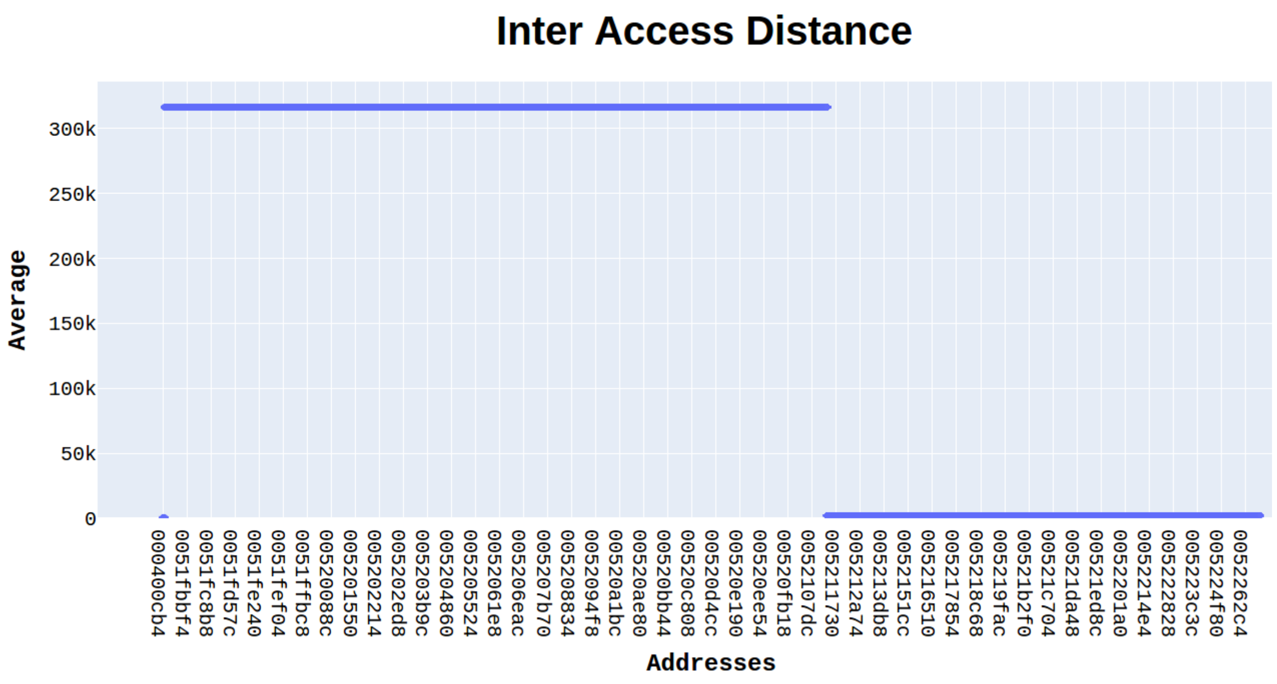


Figure 4.7: Inter access distance for 150x150 matrix multiplication.

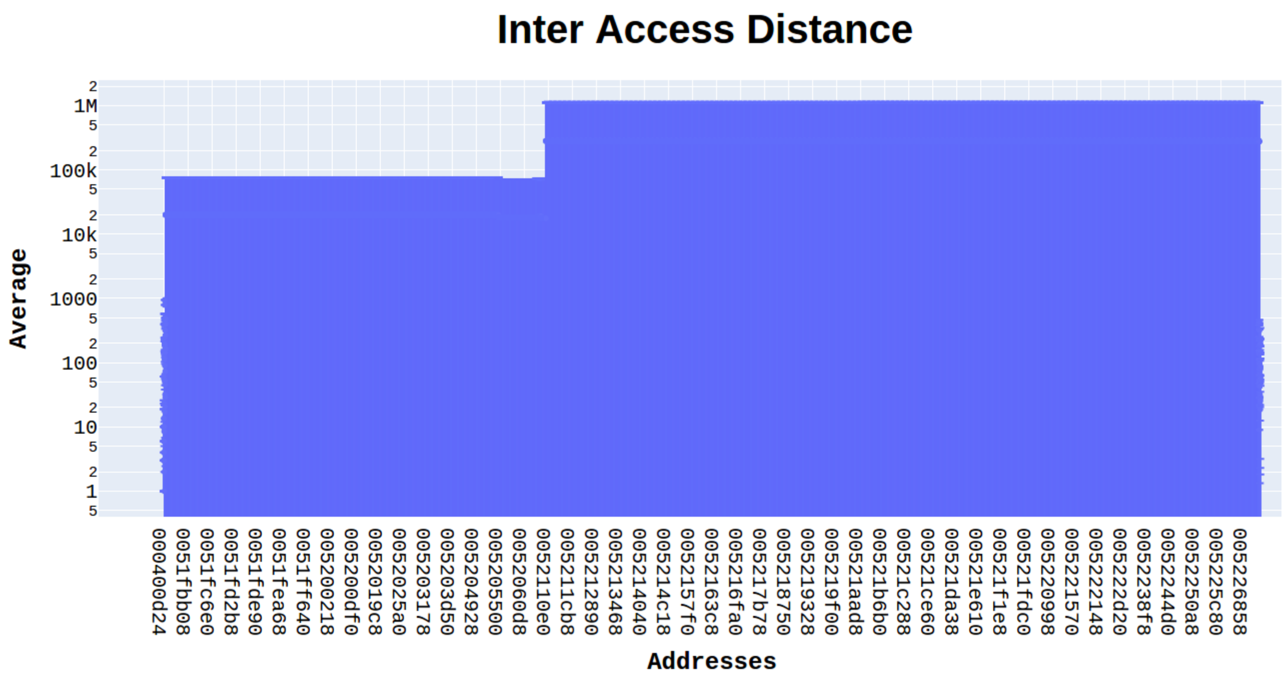


Figure 4.8: Inter access distance for 150x150 blocked matrix multiplication.

Figure 4.7 and Figure 4.8 present the inter access distance for 150x150 matrix multiplication, normal and blocked. As we can see, the number of the accesses is huge, and the error bars are distinct. Figure 4.9 and Figure 4.10 shows a zoom in the Figure 4.7 and Figure 4.8.

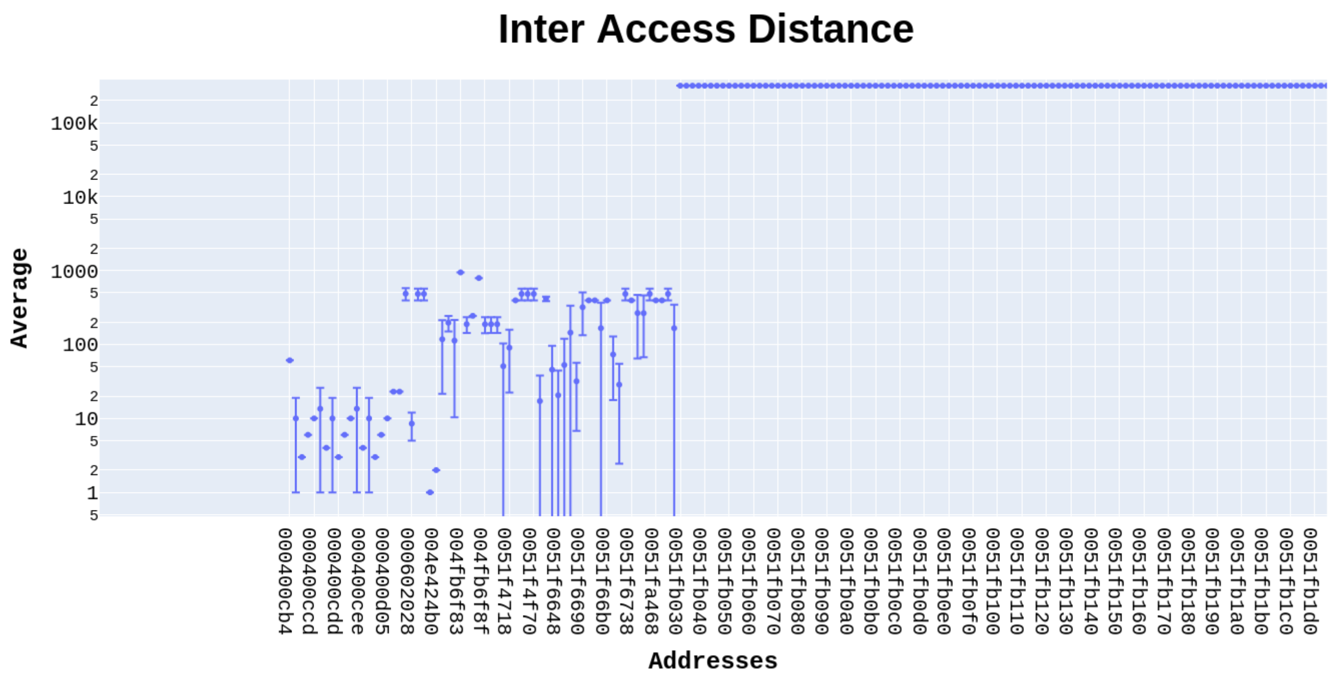


Figure 4.9: Zoom in inter access distance for 150x150 matrix multiplication.

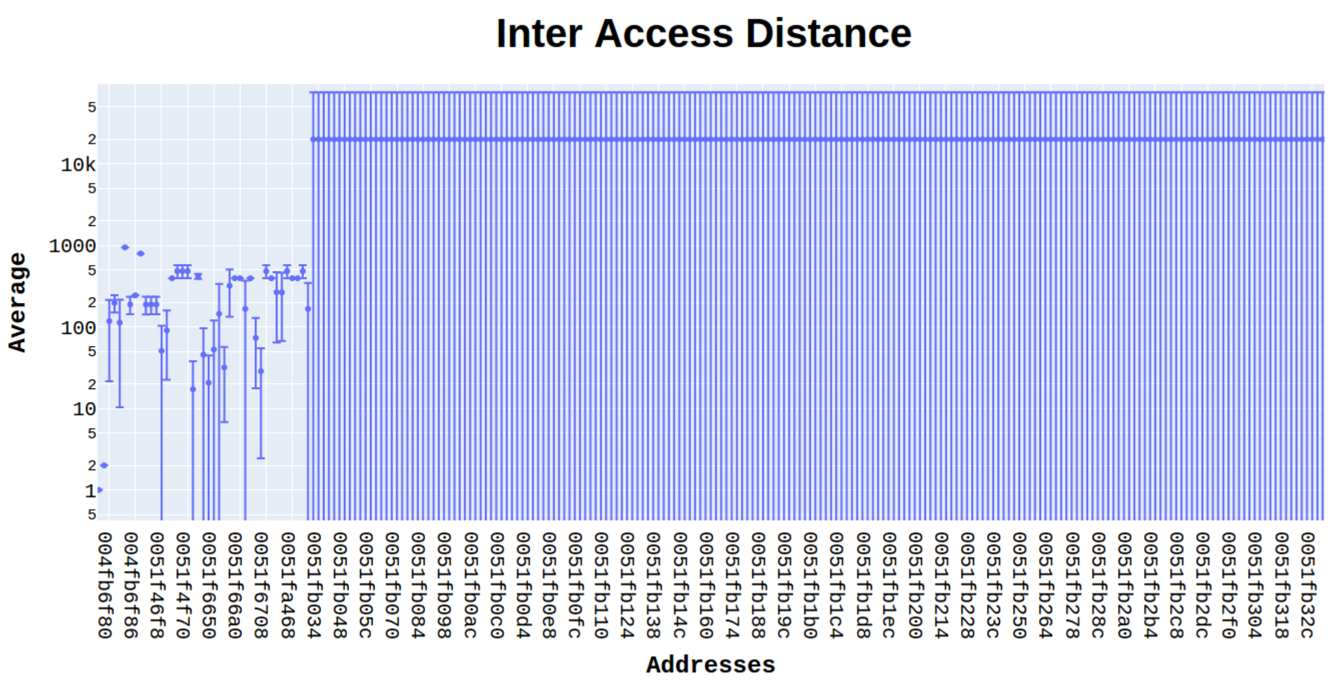


Figure 4.10: Zoom in inter access distance for 150x150 blocked matrix multiplication.

Zooming in, figures show information about every memory address clearly. For example, in Figure 4.7, for some addresses the average inter access distance seems to be equal to zero, but in Figure 4.9 the actual value of the inter access distance is shown. These values are very small compared to the rest, for this reason they can't be shown in the plot.

Comparing Figure 4.7 with Figure 4.8 we conclude in:

1. The pattern followed by the average distance of the successive memory accesses for each address differs.

2. Average inter access distance is bigger for some addresses, while source code is blocked matrix multiplication, and smaller for some others.

3. Also, standard deviation is bigger for most addresses, while source code is blocked matrix multiplication. For the first addresses that appear in the plot (left side of the plot), the average inter access distance and the standard deviation remain the same. This is explained by the fact that these accesses are related to operations that take place before the multiplication of matrices and are stable for both types of multiplication.

4.2.2 Total Accesses

In this type of visualization, we compare Figure 4.11 with Figure 4.12.

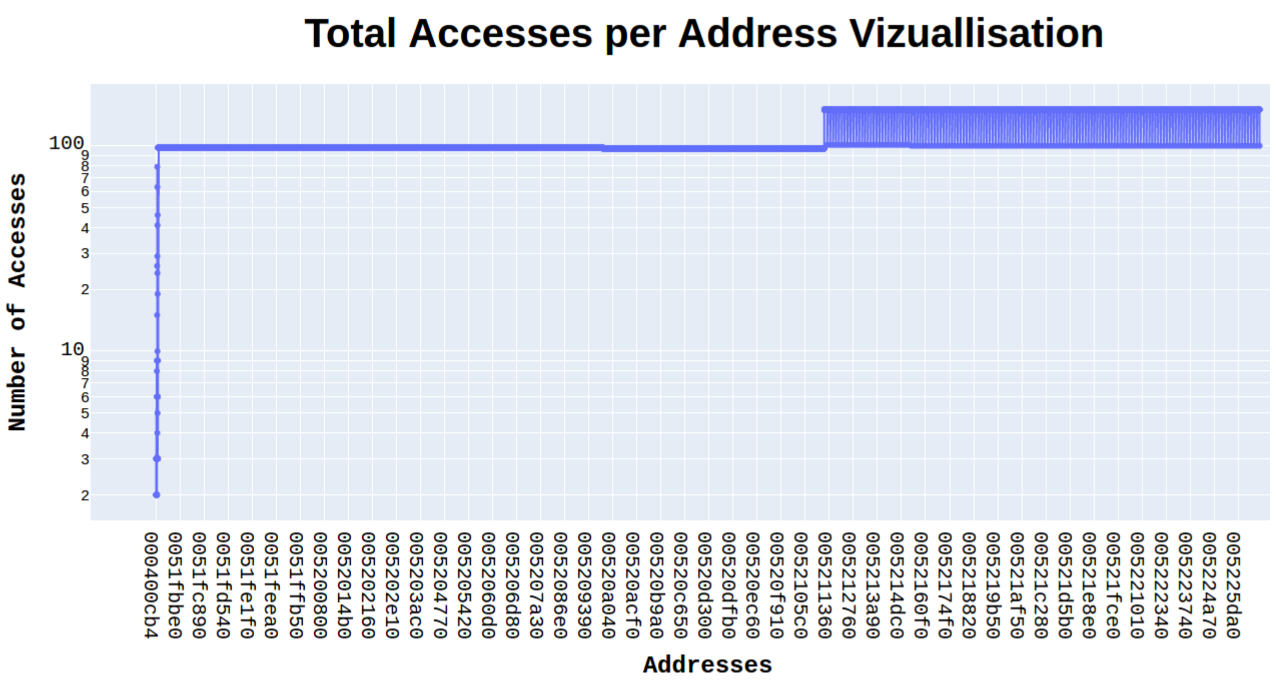


Figure 4.11: Total accesses for 150x150 matrix multiplication.

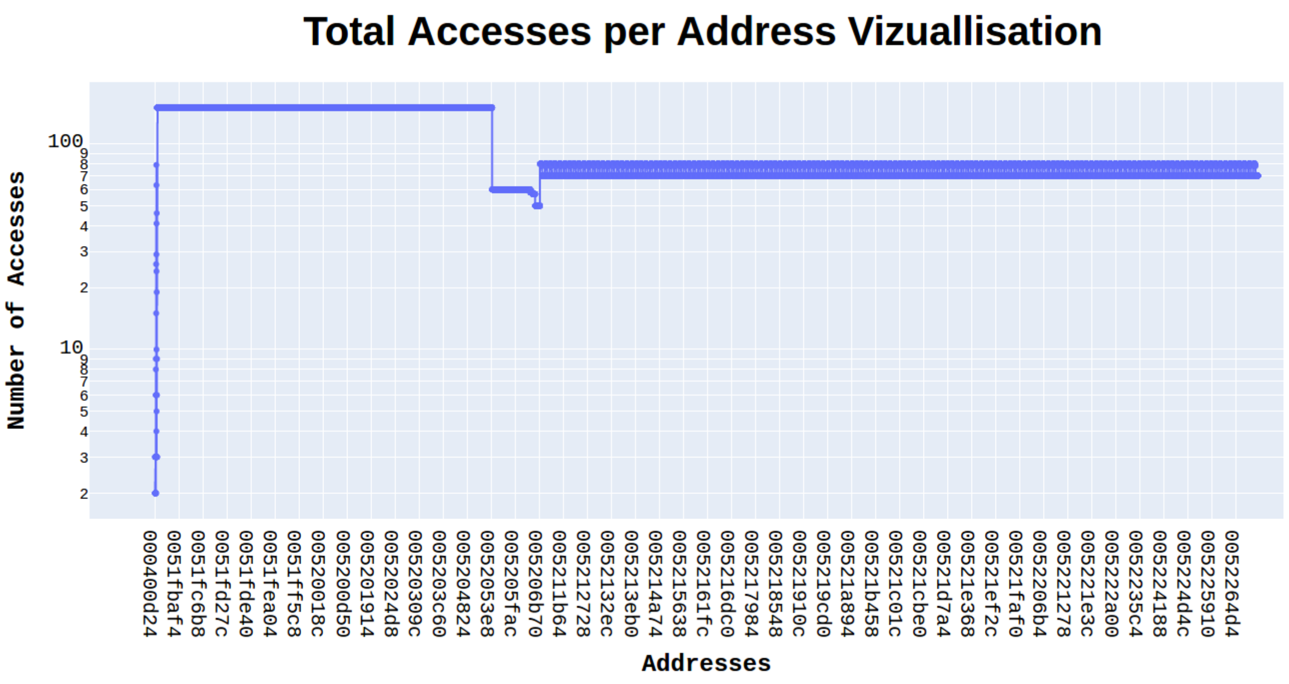


Figure 4.12: Total accesses for 150x150 blocked matrix multiplication.

We observe that total accesses follow the same pattern in both figures (source code matrix multiplication and blocked matrix multiplication). The first addresses shown on the plot have fluctuations in the access numbers. Later, for some sequential addresses, the access pattern is stable. And for the last addresses being accessed, accesses number, range between two values. Furthermore, the number of accesses for the majority of memory addresses remains the same for both source codes.

4.2.3 Timeline of Accesses

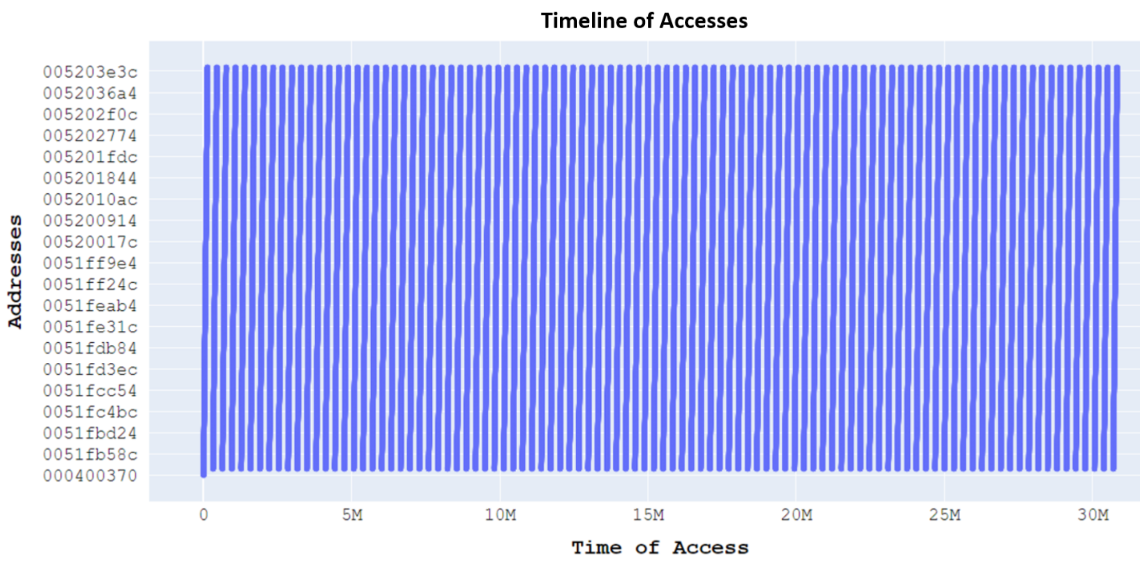


Figure 4.13: Timeline of accesses for 150x510 matrix multiplication.

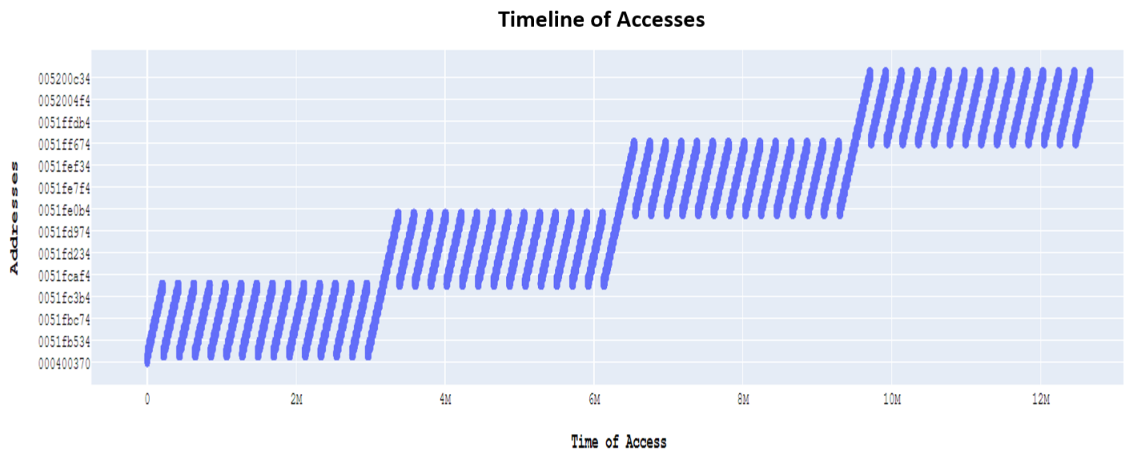


Figure 4.14: Timeline of accesses for 150x510 blocked matrix multiplication.

We can understand better the memory access pattern by observing Figure 4.13 and Figure 4.14. These Figures provide us with information about the locality of the addresses the program used. In blocked matrix multiplication, accesses to specific addresses occur between shorter intervals. As we see, data locality is increased while source code is blocked matrix multiplication. Accesses in successive addresses occur in smaller time gap than with the other source code.

Chapter 5

Conclusion

The growing speed gap between the CPU and main memory has created the need for better usage of the memory [15]. The ability to properly utilize the memory subsystem is critical for memory performance acceleration. This Thesis, provides a tool that can detect and visualize the memory access patterns, increasing application memory efficiency [16].

5.1 Summary and Conclusions

In this Thesis, we designed a tool to extract memory accesses from trace files generated using Gleipnir, a plug in built for Valgrind. After taking the memory trace, we implemented several procedures to create the data structures we needed for the visualizations. First, we decreased the size of the trace file by deleting lines with unnecessary information. After that we created the functions stack, as analyzed in Chapter 3. Finally, we created a GUI for our tool and we gave some examples of the tool's functionality.

The tool's functionality can easily be seemed by using it to visualize memory access patterns for a source code. The graphs our tool creates, provide information about the memory use. The graphs show which are the ”hot” areas of the memory, how often a memory address is being accessed, and which pattern the total memory accesses follow during the execution of the program. Having this information, developers can overcome, to some extent, the ”Memory Wall” [17].

5.2 Future Work

Because Gleipnir creates huge files, there is a need for optimizations in Gleipnir tool, to create smaller data files. The functions that our tool provides, handle big amounts of data, and utilize a large amount of memory similarly. Rewriting sections of the code in C will assist decrease the tool's memory footprint.

Furthermore, our tool allows developers to focus on a specified functions that cause memory accesses. An interesting expansion of our tool, will be the option of visualizing graphs for specific data structures, like arrays or variables. Other parts that can be optimized is the GUI, by providing additional choices for different kinds of visualizations and a command-line interface (CLI).

Bibliography

[1] Philip Machanick. Approaches to addressing the memory wall. Technical Report QLD 4072, School of IT and Electrical Engineering, University of Queensland Brisbane, Australia, 2002.

[2] Example of good locality. `https://www.eetimes.com/optimizing-for-instruction-caches-part-1/`. Date of access: 08-07-2021.

[3] Peter Grun, Nikil Dutt, and Alex Nicolau. Apex: Access pattern based memory architecture exploration. In *Proceedings of the 14th International Symposium on Systems Synthesis*, ISSS '01, page 25–32, New York, NY, USA, 2001. Association for Computing Machinery.

[4] Christos Ntogkas. Design and implementation of a tool for memory access pattern visualization. Diploma Thesis, University of Thessaly Department of Electrical and Computer Engineering, 2007.

[5] Locality of reference. `https://en.wikipedia.org/wiki/Locality_of_reference`. Ημερομηνία πρόσβασης: 06-7-2021.

[6] Spencer W. Ng Bruce Jacob and David T. Wang. *Memory Systems*. Morgan Kaufmann, San Francisco, 1 edition, 2008.

[7] Matthew Naylor and Colin Runciman. *Implementation and Application of Functional Languages*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1 edition, 2008.

[8] Linear memory access pattern. `https://en.wikipedia.org/wiki/Memory_access_pattern`. Ημερομηνία πρόσβασης: 06-7-2021.

[9] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[10] Tomislav Janjusic and Krishna Kavi. Gleipnir: A memory profiling and tracing tool. *SIGARCH Comput. Archit. News*, 41(4):8–12, December 2013.

[11] Plotly documentation. `https://plotly.com/python/`. Ημερομηνία πρόσβασης: 06-7-2021.

[12] Standard deviation. `https://en.wikipedia.org/wiki/Standard_deviation#cite_note-StatNotes-1`. Ημερομηνία πρόσβασης: 09-07-2021.

[13] Jeanne Ferrante. *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA, 2007.

[14] Shipman, john w. (2010-12-12), tkinder referance. `http://alexandre.benoit.83.free.fr/archives/1415/adumas/ISN/tkinter.pdf`. Date of access: 08-07-2021.

[15] Margaret Martonosi, Anoop Gupta, and Thomas Anderson. Memspy: Analyzing memory system bottlenecks in programs. *SIGMETRICS Perform. Eval. Rev.*, 20(1):1–12, June 1992.

[16] Byunghyun Jang, Dana Schaa, Perhaad Mistry, and David Kaeli. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):105–118, 2011.

[17] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.