View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Thessaly Institutional Repository

Department of Electrical and Computer Engineering
University of Thessaly

Design and implementation of a tool for memory access pattern visualization

By

Christos Ntogkas

Supervisors
Christos D. Antonopoulos, Assistant Professor
Spyros Lalis, Associate Professor

Volos 2017,Greece

Title of Thesis: Design and implementation of a tool for memory access pattern visualization

(Σχεδίαση και υλοποίηση εργαλείου για την οπτικοποίηση του προτύπου προσπέλασης στη μνήμη)

Author: Christos Ntogkas / Χρήστος Ντόγκας

Supervisors:
Christos D. Antonopoulos, Assistant Professor
Spyros Lalis, Associate Professor

University of Thessaly
Department of Electrical and Computer Engineering

Volos 2017, Greece

Dedicated to my family and friends.

ACKNOWLEDGEMENTS

I would like to thank my supervisors Christos D. Antonopoulos and Spyros Lalis for their help and guidance throughout this work. It is due to their inspiration and continuous encouragement that I was able to successfully complete this work. I would also like to thank my family and friends for their continuous support throughout these years.

Περίληψη

Όσο οι βελτιώσεις στην επεξεργαστική ισχύ συνεχίζουν να υπερτερούν των βελτιώσεων στις επιδόσεις της μνήμης, σε συνδιασμό με την αμέλεια των προγραμματιστών για τις ιδιότητες της μνήμης και την ιεραρχία της, εργαλεία για την κατανόηση της συμπεριφοράς προσπέλασης μνήμης είναι αναπόφευκτα ζωτικής σημασίας για τη βελτιστοποίηση της εκτέλεσης εφαρμογών που χειρίζονται μεγάλο όγκο δεδομένων.Τα πρότυπα προσπέλασης μνήμης, πρότυπα σύμφωνα με τα οποία ένα σύστημα ή ένα πρόγραμμα διαβάζει και γράφει στη μνήμη, διαφέρουν στο επίπεδο τοπικότητας αναφοράς και επηρρεάζουν δραστικά τις επιδόσεις του. Η αναγνώριση των προτύπων πρόσβασης και η χρήση αυτής της γνώσης για την οργάνωση των λειτουργειών και την επιλογή κατάλληλων δομών δεδομένων ώστε να γίνει ο κώδικας πιο φιλικός προς την μνήμη cache μπορεί να οδηγήσει σε σημαντική επιτάχυνση της εκτέλεσης του προγράμματος.

Ο σκοπός αυτής της διπλωματικής είναι ο σχεδιασμός και η υλοποίηση ενός εργαλείου ανάλυσης της συμπεριφοράς των εφαρμογών στη μνήμη και οπτικοποίησης των προτύπων προσπέλασής τους σε αυτή. Το προτεινόμενο εργαλείο προσφέρει μια νέα οπτικοποίηση της συμπεριφοράς της προσπέλασης μνήμης, επικεντρώνοντας στην κατανομή των προσπελάσεων ανά συνάρτηση, νήμα και δομή δεδομένων. Αυτού του είδους η οπτικοποίηση επιτρέπει στον προγραμματιστή να κατάλαβει με ευκολία γιατί παρουσιάζονται προβλήματα με τις επιδόσεις και προσφέρει βοήθεια στην αναμόρφωση των δεδομένων και του κώδικα.

Abstract

As improvements in processing power continue to outpace improvements in memory performance, combined with the developers' disregard for memory properties and hierarchy, tools to understand memory access behavior are inevitably vital for optimizing the execution of data-intensive applications. Memory access patterns, the patterns with which a system or a program reads and writes to memory, differ in the level of locality of reference and drastically affect performance. Identifying access patterns and using this knowledge to organize I/O operations and choose appropriate data structures so as to make the code cache-friendly can provide a significant speedup to the execution of the program.

The purpose of this thesis is to design and implement a tool for analyzing the memory behavior of applications and visualizing their access patterns. The tool we introduce provides a new visualization of the memory access behavior, focusing on the distribution of accesses by function, thread and structure. Such visualization allows the developer to easily understand why performance issues occur and provides assistance in refactoring data and code.

Keywords: Memory trace, Memory access pattern, Profiling, Visualization, Data locality.

Contents

1. Introduction....9
   1.1 Problem description....9
   1.2 Thesis structure....10
2. Background....11
   2.1 Memory hierarchy....11
   2.2 Locality of reference....12
   2.3 Memory access patterns....13
3. Implementation....17
   3.1 Memory trace....17
   3.2 Data filtering and organization 20
   3.3 Data processing....21
   3.4 Visualization of results....23
   3.5 User interface....23
4. Evaluation....26
   4.1 Matrix multiplication....26
   4.2 Mandelbrot....31
   4.3 Bzip2....33
5. Related Work 38
6. Conclusion....39
   6.1 Future work....39
   Bibliography....41

List Of Figures

Figure 1: Performance increase gap between processor and memory....9
Figure 2: Memory hierarchy....11
Figure 3: Example of good locality....12
Figure 4: Example of bad locality....13
Figure 5: Sequential memory access....14
Figure 6: Random memory access....14
Figure 7: Strided memory access....14
Figure 8: Scatter memory access....15
Figure 9: Gather memory access....15
Figure 10: System overview....17
Figure 11: Example source code....19
Figure 12: Gleipnir trace file example....19
Figure 13: Data filtering and organization flowchart....21
Figure 14: Graphical user interface....24
Figure 15: Matrix multiplication reuse distance analysis graph....26
Figure 16: Matrix multiplication variable metrics graph....27
Figure 17: Matrix multiplication memory activity graph....28
Figure 18: Matrix multiplication memory activity graph (zoomed in)....28
Figure 19: Matrix multiplication execution statistics....29
Figure 20: Matrix multiplication cache statistics....29
Figure 21: Blocked matrix multiplication memory activity graph....30
Figure 22: Blocked matrix multiplication average reuse distance comparison....30
Figure 23: Mandelbrot reuse distance analysis graph....31
Figure 24: Mandelbrot variable metrics graph....31
Figure 25: Mandelbrot memory activity graph....32
Figure 26: Mandelbrot memory activity graph (zoomed in)....32
Figure 27: Mandelbrot execution statistics....33
Figure 28: Mandelbrot cache statistics....33
Figure 29: Bzip2 (compression) reuse distance analysis graph....34
Figure 30: Bzip2 (decompression) reuse distance analysis graph....34
Figure 31: Bzip2 (compression) variable metrics graph....35
Figure 32: Bzip2 (decompression) variable metrics graph....35
Figure 33: Bzip2 (compression) memory activity graph....36
Figure 34: Bzip2 (decompression) memory activity graph....36

List Of Tables

Table 1: Algorithms and trace file sizes..20

1

Introduction

1.1 Problem Description

Memory is a major performance bottleneck in embedded and high-performance computing systems. The decreasing cost of memory and storage and the increasing size of it has led to a large growth in the volume of data handled by applications. This rapid growth in main memory has practically phased out disk I/O even in data-intesive applications [1], leading to memory performance taking the main focus in optimization. Nowadays, any kind of data processing, collection, analysis and even service virtualization are all contenders for memory utilization.

For the past 20 years there has been a disproportionate increase in performance between processors and memory [2] (fig. 1),a trend that does not appear to be changing in the near future [3][4].The processor speed increase has greatly surpassed latency gains to main memory. While bandwidth to main-memory has increased substantially by using wider and multi-channel buses, the latency did not enjoy a similarly dramatic reduction. This is something that concerned researchers and system architects for over 20 years and is also known as the Memory Wall [5].

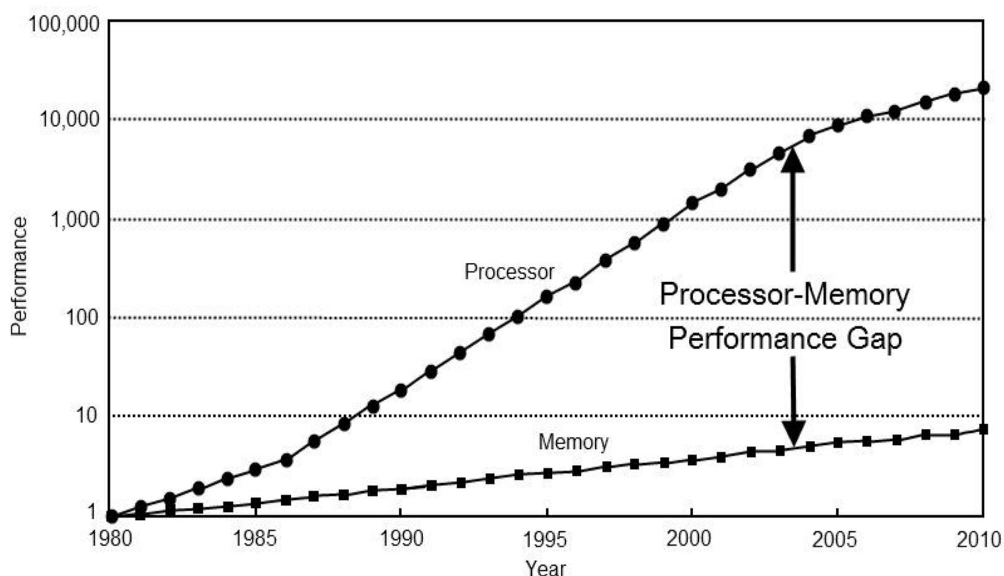

Figure. 1 Performance increase gap between processor and memory.

In order to hide this latency, processors use a complex, hierarchical multi-layered sub-system for accessing memory. The pattern of these accesses drastically affects performance [6], has implications for the exploitation of parallelism [7] and distribution of workload in shared memory systems [8] as well as for security [9][10]. Given the impact memory access patterns have on the performance of a system or program it becomes clear that understanding, analysis and improvement of these patterns is imperative.

The focus of this thesis will be the design and implementation of a tool that offers software developers and researchers the ability to visualize and understand these patterns in order to optimize memory performance.

1.2 **Thesis Structure**

This thesis is structured in three parts. The first part includes Chapter 2 which discusses background information about the memory in a system. Specifically, in section 2.1 , we analyze the memory hierarchy , in section 2.2 we describe data locality and finally in section 2.3 we examine memory access pattern and its effects in a system or program.

The second part includes Chapter 3 and Chapter 4. In Chapter 3 we provide details about the different stages of the tool's execution and the design and implementation of each of them. In Chapter 4 we validate the tool and demonstrate its functionality using with various source codes.

The last part includes Chapter 5 and Chapter 6. Chapter 5 presents related work focusing on tools for memory profiling and analysis of access patterns. Finally, we summarize and conclude this thesis in Chapter 6, and propose some additions to the tool and potential directions for future research.

2

Background

2.1 **Memory Hierarchy**

Programmers and system designers, over the years, want an ever increasing amount of memory with low latency but the additional size and speed come costly. Multilevel caching provides the illusion of large, fast and cheap memory.

The performance of the memory hierarchy is a prime optimization goal in computer architectural design. Moreover, software performance prediction and optimization is based on the analysis of the interaction of the code with the memory. Figure 2 depicts a typical memory organization (layers, technology used and associated tradeoffs).

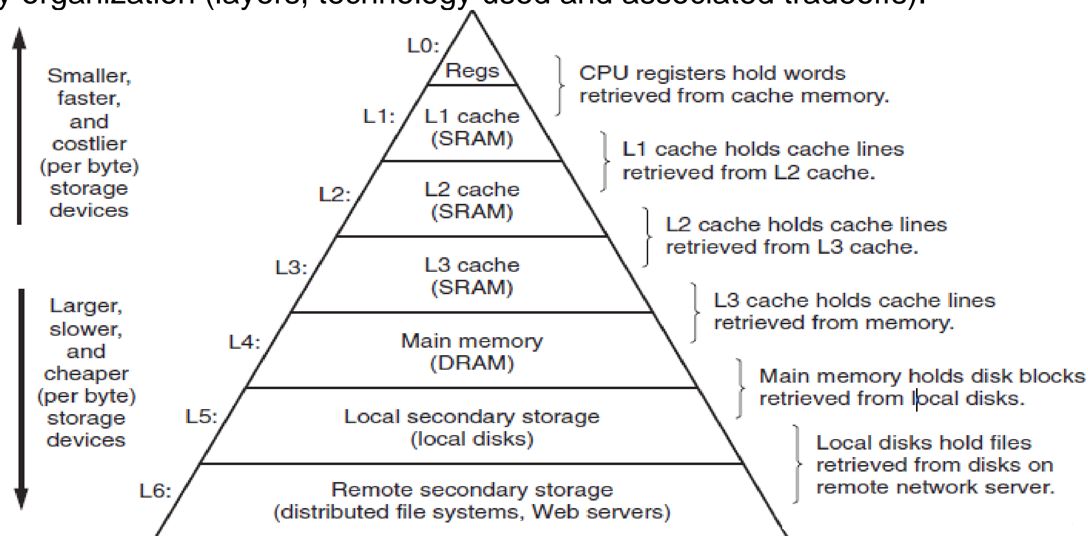

Figure. 2 Memory hierarchy. Figure from Randal E. Bryant and David R. O'Hallaron. 2010. Computer Systems: A Programmer's Perspective (2nd ed.).Addison-Wesley Publishing Company, , USA.

When designing for high performance it is necessary to take into account the restrictions of the memory hierarchy such as the size and capabilities of each layer. Programmers that have a deep understanding on the interaction of their code with the memory system can write the code in a way that optimally exploits the memory hierarchy [12].

2.2 **Locality of reference**

During execution of a program it is a common occurrence that the same memory cells are accessed repeatedly or that memory cells near a previously accessed cell are the next to be accessed. This phenomenon is called locality of references and is the main reason why caches are effective [13].

Increase in locality will improve the efficiency of the cache. The basic types of reference locality are temporal and spatial. Temporal locality refers to the reuse of specific data within a relatively small time window. The most common example of temporal locality in a program are loops which fetch the same instructions and sometimes data multiple times. Spatial locality refers to the use of data elements within relatively close storage locations. A good example of this type of locality are arrays of data which are accessed sequentially. Both of these types of locality of references are obvious in figure 3.

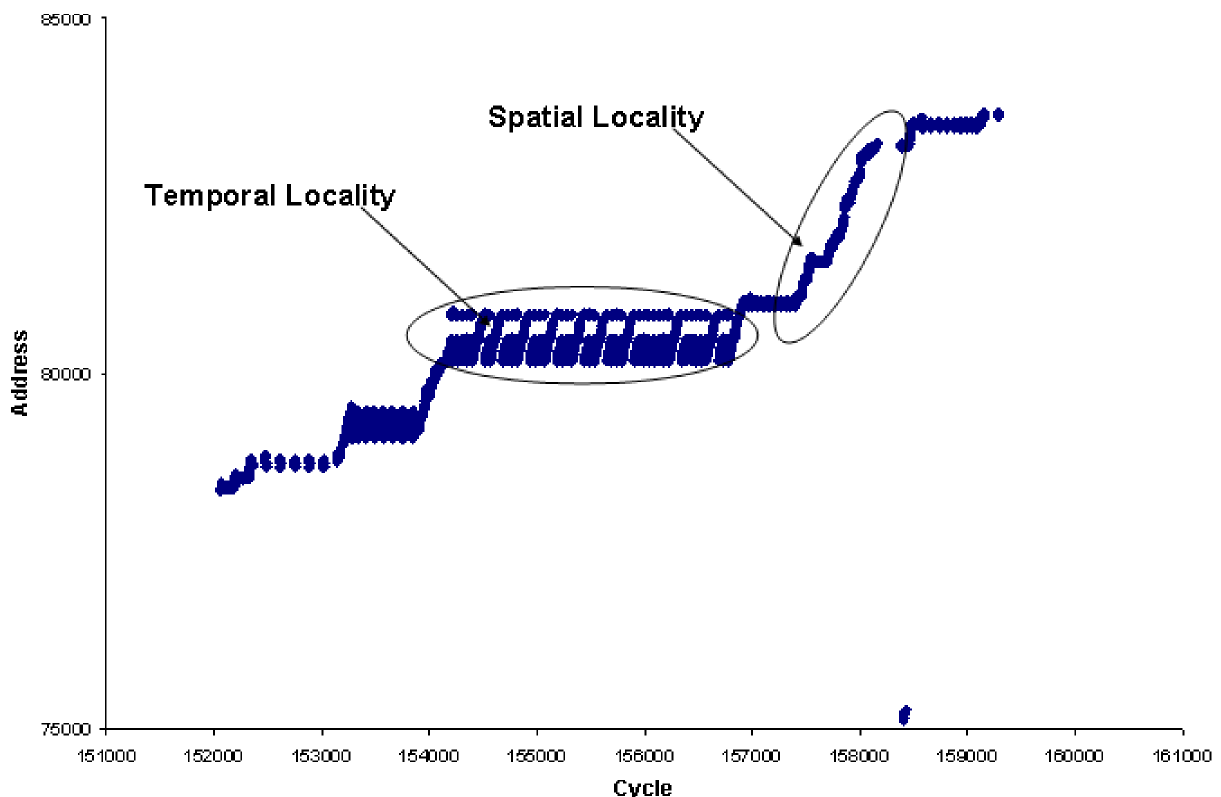

Figure. 3 Example of good locality. Figure from Optimizing for instruction caches part 1, Amir Kleen, Livadariu Mircea, Itay Peled, Erez Steinberg, Moshe Anschel, Freescale 2007.

On the other hand in figure 4 we can see an example of bad locality. A wide range of addresses are being accessed not in any particular order and there is also temporal reuse after a large number of cycles, both facts consistent with inefficient cache behavior.

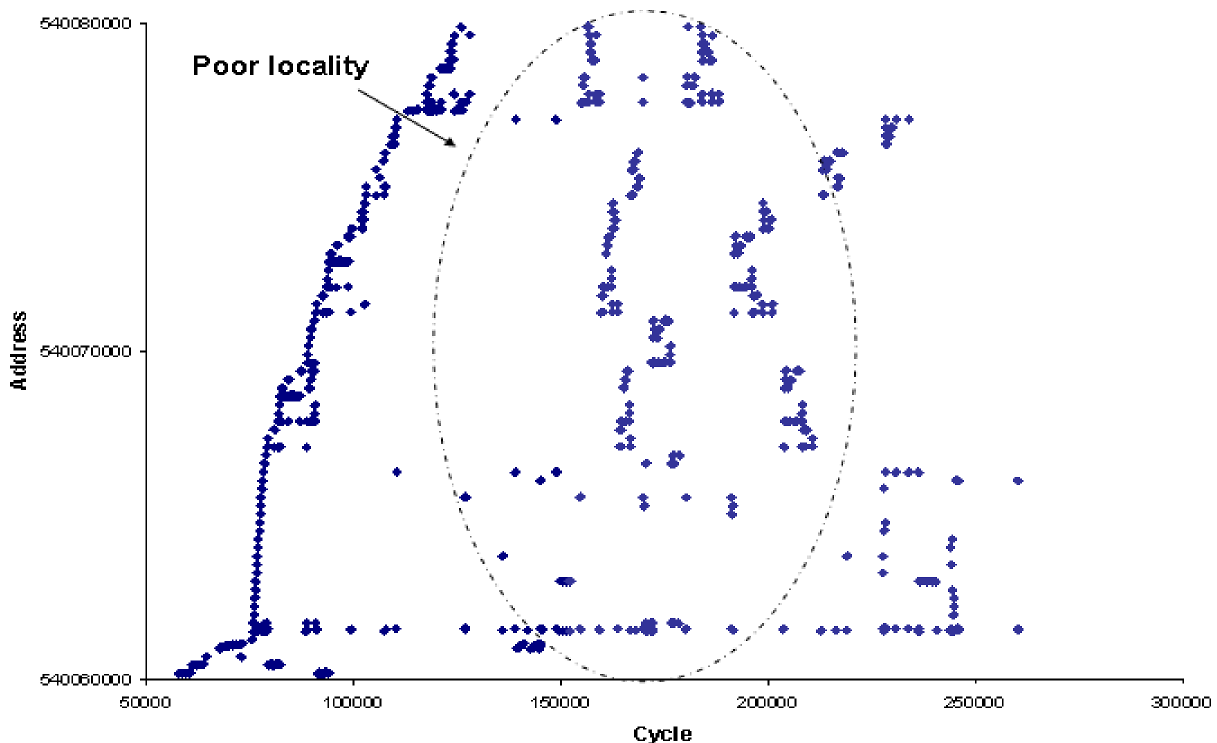

Figure. 4 Example of bad locality. Figure from Optimizing for instruction caches part 1, Amir Kleen, Livadariu Mircea, Itay Peled, Erez Steinberg, Moshe Anschel, Freescale 2007.

To go one step further, a lot of research has been done in improving cache efficiency even more. Notable solutions are increasing cache line size, while making sure that the extra space in the cache line is not wasted as well as predicting future accesses and loading data before they are needed by using a hardware prefetcher [14][15][16].

The extend and type of locality is an application-specific characteristic. Strong locality of reference is a good indicator of eligibility for performance optimization.

2.3 **Memory access pattern**

Today, due to the advancements in circuit design and emerging technologies in lithography the main performance bottleneck are memory operations and especially to secondary memory, also called the “Von Neumann bottleneck” [17]. Computer memory is usually described as 'random access', but traversal by software will still exhibit patterns that can be exploited for efficiency. While various software show similarities in structure and functionality their memory access pattern can be very different. Practically, the number of memory access patterns is unlimited [18]. There are a few memory access patterns though that tend to appear frequently. The most notable ones are:

- Sequential

The simplest access pattern, where data in consecutive memory positions are read or written in sequence.

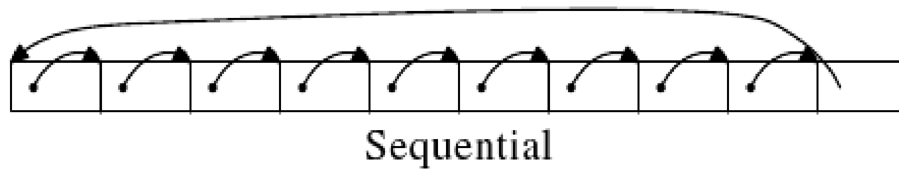

Figure. 5 Sequential memory access

- Random

Random memory access patterns are harmful to cache and memory performance. In this context , randomness is not defined so much as completely arbitrary addressing but accesses in general that are not sequential, and not to the same data or close to data that has been recently used, and not obeying a predictable pattern which be identified and exploited by the hardware prefetcher [19].

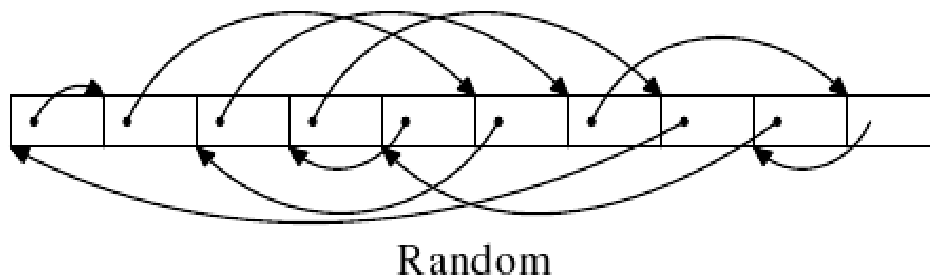

Figure. 6 Random memory access.

- Strided

Strided access pattern with stride K means accessing every Kth memory element. Stride equal to 1 is equivalent to sequential access.

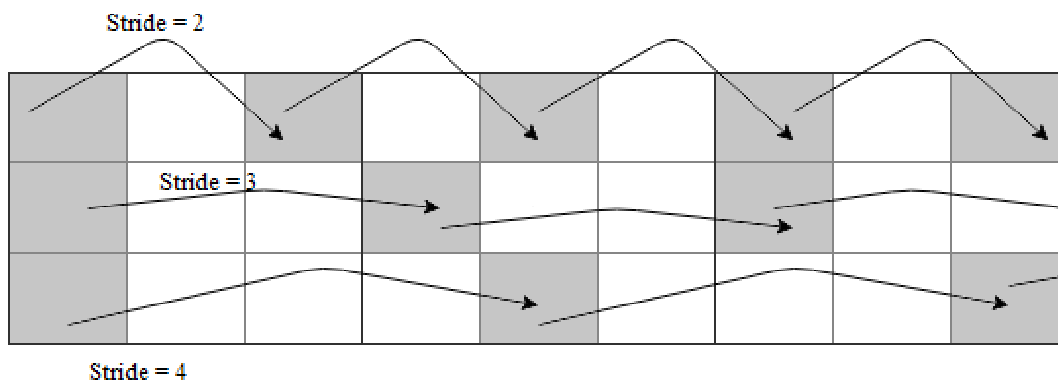

Figure. 7 Strided memory access.

- Scatter

A scatter memory access pattern combines sequential reads with indexed/random addressing for writes [20]. Compared to gather, it may place less load on a cache hierarchy since a processing element may dispatch writes in a 'fire and forget' manner

(bypassing cache altogether). However it may be harder to parallelise since there is no guarantee the writes are not indepedendent.

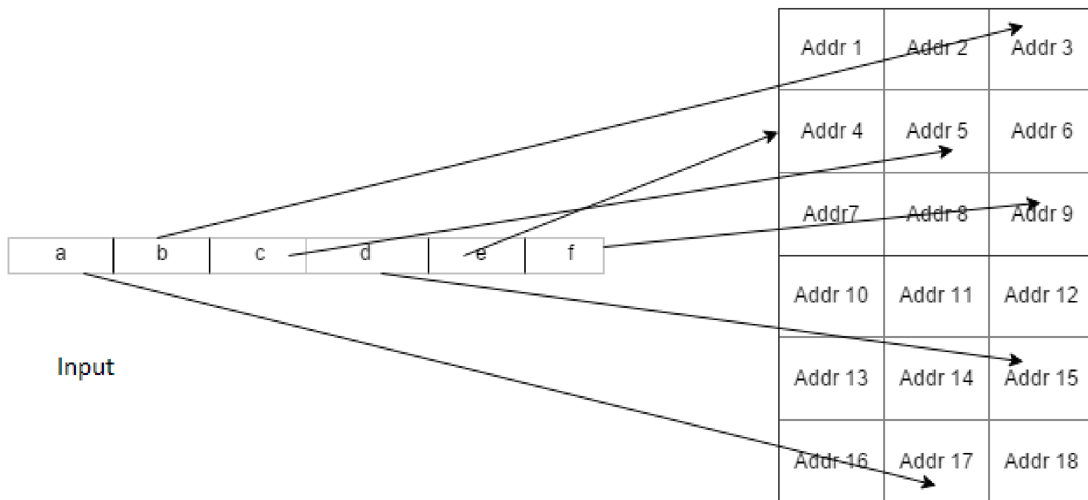

Figure. 8 Scatter memory access.

- Gather

In gather access pattern, reads are randomly addressed or indexed, whilst the writes are sequential [20]. Compared to scatter, the disadvantage is that caching is now essential for efficient reads of small elements, however its is easier to parallelise since the writes are guaranteed to not overlap.

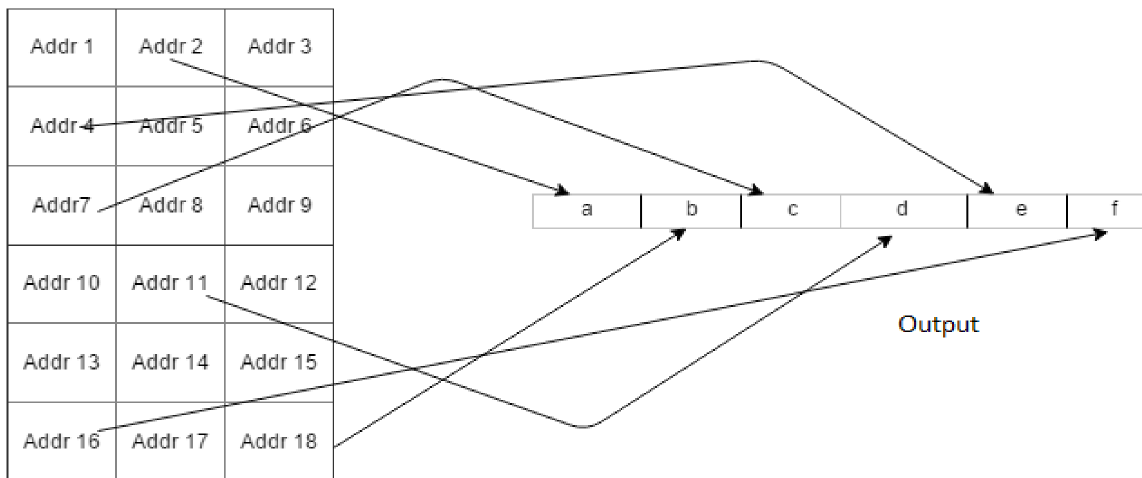

Figure. 9 Gather memory access.

Combinations of these patterns exist as well.

As we mentioned earlier, in order to achieve high performance one needs to consider the interaction of software with the memory hierarchy [21]. A step in that direction are cache-oblivious algorithms, an idea that has been around since 1996 [22][23].This type of algorithms is designed to take advantage of a CPU cache without having the size of the cache (or the length of the cache line) as an explicit parameter, thus performing well

,without modification, on a memory hierarchy with different levels of cache having different sizes. The disadvantages of these algorithms is that they are complex and limiting to the problem-solving approaches.

When designing data structures and algorithms cache hierarchy, data locality and access patterns are important parameters for memory performance optimization. While the cache hierarchy may be known beforehand, the access pattern changes, at varying degrees, even with subtle changes to the program. Even though we can try to predict the access patterns [24][25][26], the preferred choice is to analyze the current memory access patterns and then apply the proper optimizations.

With that in mind, our goal is to create a tool that will capture all memory accesses during the execution of a program and visualize them in a meaningful way so as to be able to draw conclusions on the memory access pattern of the program and potential optimization opportunities.

3

Implementation

In this chapter we first present a basic overview of the whole system and then discuss each part of the tool in more detail. Figure 10 provides a schematic representation of the system.

Figure. 10 System overview.

The functionality of the tool we introduce can be divided into four parts:

- Acquisition of the memory trace of the program to be analysed
- Filtering of the acquired data and storage into appropriate data structures
- Processing of the data
- Visualization of the results

3.1 Memory trace

As we mentioned before, memory performance has taken the center stage in optimization. This has led to a rise in the development of tools for profiling and analyzing applications. A common method followed by these tools is to observe the code of an application during its execution and collect information about its inner workings, like number of branches, function calls, CPU cycles etc. Specifically for memory behavior, there are three methods to do that, first by inserting code at compile time capable of collecting the required information, called static instrumentation, second by inserting code after the compilation that examines the instructions and collects the required data, called

dynamic instrumentation and last by simulating the memory hierarchy. The tool that we are going to use for acquiring the memory traces is called Gleipnir [27] and is a plug-in built for the popular binary instrumentation tool Valgrind [28].

There are plenty of instrumentation frameworks capable of capturing memory accesses. Some of the reasons why we chose Valgrind are:

- It is open-source and easily-customizable.
- It has been ported to a variety of platforms like AMD64,x86 and ARM.
- It is compatible with GCC.
- It widely used by the HPC community.
- Most importantly, it is the framework that Gleipnir is based upon.

After doing research we came to the conclusion that Gleipnir was a good choice for our tool because of its ability to associate dynamically allocated blocks of memory with data structures in the source code of the program as well as attributing the accesses to the specific variables,functions or threads that caused the access[27].

Gleipnir uses the dynamic instrumentation method , which although adds a significant runtime overhead,making execution tens or hundreds of times slower, it provides more details and more flexibility than the alternatives. It operates on instruction events coming from Valgrind's internal debug parser. The debug parser and the debug information of the instructions are used to relate the various accesses to the memory objects causing them.The dynamic blocks of memory are intercepted and recorded using wrappers for the malloc() library. Figures 11 and 12 display an example source code and trace file produced by Gleipnir.

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

struct typeA GlStrc;
struct typeA glStrcArray[10];

int glScalar;
int glArray[10];

void func(struct typeA StrcParam[])
{
  int i;

  for(i=0; i<3; i++){
    glStrcArray[i].var1 = glScalar;
    glStrcArray[i].myArray[0] = glArray[0];
    StrcParam[i].var1 = glArray[i];
  }
  return;
}

int main(void)
{
  GL_START_INSTR;

  struct typeA lcStrcArray[5];
  int i, lcScalar, lcArray[10];

  glScalar = 10;
  lcScalar = 20;

  for(i=0; i<2; i++)
    lcArray[i] = glScalar;

  func(lcStrcArray);

  GL_STOP_INSTR;

  return 0;
}
```

Figure. 11 Example source code.

```
START PID 517
X THREAD_CREATE 0:1
S 7ff000490 8 1 S main LV _zzq_result
L 7ff000490 8 1 S main
S 000601070 4 1 G main GV glScalar
S 7ff00049c 4 1 S main LV lcScalar
S 7ff000498 4 1 S main LV i
L 7ff000498 4 1 S main LV i
L 000601070 4 1 G main GV glScalar
L 7ff000498 4 1 S main LV i
S 7ff000460 4 1 S main LS lcArray[0]
M 7ff000498 4 1 S main LV i
L 7ff000498 4 1 S main LV i
L 000601070 4 1 G main GV glScalar
L 7ff000498 4 1 S main LV i
S 7ff000464 4 1 S main LS lcArray[1]
M 7ff000498 4 1 S main LV i
L 7ff000498 4 1 S main LV i
S 7ff000330 8 1 S main
S 7ff000328 8 1 S func
S 7ff000310 8 1 S func LV StrcParam
S 7ff000324 4 1 S func LV i
L 7ff000324 4 1 S func LV i
L 000601070 4 1 G func GV glScalar
L 7ff000324 4 1 S func LV i
S 000601080 8 1 G func GS glStrcArray[0].var1
L 000601260 4 1 G func GS glArray[0]
L 7ff000324 4 1 S func LV i
S 000601088 4 1 G func GS glStrcArray[0].myArray[0]
L 7ff000324 4 1 S func LV i
L 7ff000310 8 1 S func LV StrcParam
L 7ff000324 4 1 S func LV i
L 000601260 4 1 G func GS glArray[0]
S 7ff000340 8 1 S func LS lcStrcArray[0].var1
M 7ff000324 4 1 S func LV i
L 7ff000324 4 1 S func LV i
L 000601070 4 1 G func GV glScalar
L 7ff000324 4 1 S func LV i
S 0006010b0 8 1 G func GS glStrcArray[1].var1
L 000601260 4 1 G func GS glArray[0]
L 7ff000324 4 1 S func LV i
S 0006010b8 4 1 G func GS glStrcArray[1].myArray[0]
L 7ff000324 4 1 S func LV i
```

Figure. 12 Gleipnir trace file example

Figure from Gleipnir: a memory profiling and tracing tool, Tomislav Janjusic and Krishna Kavi, SIGARCH Comput. Archit. News 41, 4 (December 2013)

Each entry of the trace file created from Gleipnir includes the following information:

Operation Address Memory_Size Thread_Id Scope Function Variable_Info Element

- ***Operation***: L (load) / S (store) / M (modify) / I (instruction) / X (keyword) depending on the type of access. Keyword accesses are special lines inserted in the trace file by Gleipnir to describe certain events during tracing.
- ***Address***: Memory address (in the simulated address space of Valgrind).
- ***Memory_size***: Size of memory accessed.
- ***Thread_Id***: Id of the thread that caused the access.
- ***Scope***: G (global) / S (stack) / H (heap) depending on the location of the element.
- ***Function***: Name of the function call that caused the access.
- ***Variable_Info***: G (global) / L (local) and V (variable) / S(structure) depending on the

element. In the case of structure the accessed member is displayed as well.

In the current example, Gleipnir is configured to display information pertaining only to loading and storing of a variable. It has the ability , though, to intercept the accesses for instructions as well. According to its creators, Gleipnir offers much more functionality [27] which is, however, not immediately relevant to our purpose.

Gleipnir is ,however, associated to a number of drawbacks. Perhaps its biggest one is that due to the way it works, if we want to capture all the memory access of a single function call, first Gleipnir would intercept and trace all the accesses in the entire code and then ignore the unneeded information. Therefore, the trace files produced even from the simplest of codes are disproportionately large and execution is sometimes 30-100 times slower than the original source code [27]. Some typical algorithms and the trace files they produce are shown in Table 1.

Benchmark	L/S Instructions	Trace file size
basic math	651,272,648	16GB
AES	192,740,716	5.6GB
Dijkstra	166,177,792	4.5GB
FFT	130,850,859	3.2GB
Qsort	111,689,557	2.8GB

Table. 1 Algorithms and trace file sizes for the MiBench benchmark suite.

Also, up until its latest release, accesses of every thread are reported independently. In the case of interleaved execution of threads the sequence of accesses in the trace file does not correspond to the true sequence of accesses.

On the positive side, Gleipnir is actively maintained by dedicated developers so improvements,fixes and new functionality are yet to come. Being a plug-in tool for a such a widely used and modular framework as Valgrind is in itself a great quality as it can be combined with other tools for more advanced purposes.

3.2 Data filtering and organization

As stated earlier, Gleipnir produces large trace files. Since we don't have unlimited memory we need to devise a way to reduce the file size of the trace file,filter the trace data and organize them in suitable structures in order to use them further along the execution of our tool.

The first problem to tackle is the size of the trace file. In order to reduce its size we will substitute it with a lookup table and an access list both stored on binary files. We are going to parse the trace file line by line and for every unique memory access we encounter, after we check the lookup table for duplicates ,we store the memory access as a new record on a dynamically allocated lookup table. For every valid memory access in the trace file, excluding keyword accesses of type X, we are going to add a record to the access list. This simple change in the storage format ,as we will observer in Chapter 4, reduces the storage requirements of the trace file by 65-85% depending on the program.

The data, extracted from the trace file, for every valid memory access will be stored in record instances. The unique record instances will be added to the Lookup table. For every access, a pair of operation type and Lookup table index will be added to the Access List. The detailed flowchart of the data filtering and organization along with the details on the used structure types are shown in figure 13.

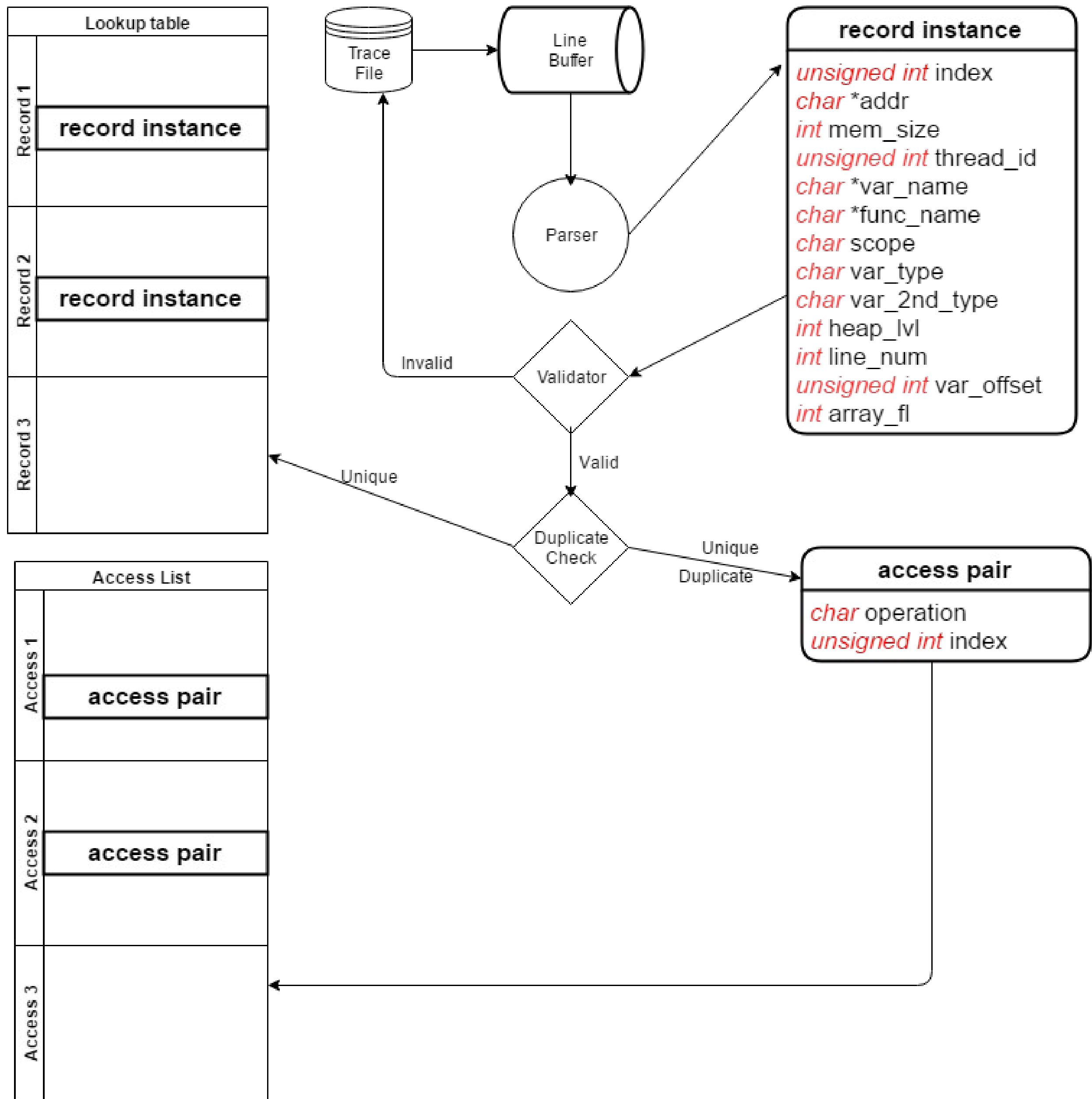

Figure. 13 Data filtering and organization flowchart.

3.3 **Data processing**

At this point we have filtered out the unwanted data and organized the rest into useful structures. We are now ready to process these structures in order to get meaningful

information about the program's access pattern. Based on the analysis we want to perform, we will focus on the following metrics to provide us with a detailed insight of the characteristics of the memory access pattern of the program:

- **Reuse distance**

  Data reuse is an intrinsic property of each program and as such it s not directly connected to machine architecture or memory parameters. This property allows for a clearer picture of the program's locality and identification of the most frequently used as well as the most sparsely accessed data [30].

  Two common techniques for calculating the reuse distance of data are exact reuse-distance measurement and statistical prediction of locality in all executions based on the analysis of a few executions. We decided to go with the exact reuse-distance measurement method to get the most accurate depiction of the program's use of data. For each data access, the reuse distance is the number of distinct data elements accessed between the current and previous accesses to the same element. Using the Access list and Lookup table arrays we can find the order of the memory access and calculate the reuse distance for every data element.

  Identifying the data reuse frequency can help organize the program's code and data structures in a more cache friendly manner, lessening the burden imposed by wasteful memory accesses and ultimately improving performance [31]. The average reuse distance is another useful statistic as it gives us a picture of the program's locality.

- **Variable usage**

  Depending on the program in question, the usage of variables and structures may vary greatly. Image processing tends to reuse data a lot more than sorting arrays or manipulating files for example. Detailed information about the variables / structures accessed and the size of memory modified , in experienced eyes , can often reveal much insight on the functionality of the program.

  Storing the memory size of each access in the Lookup table and all the access pairs in the Access list array provides us ,at any point during execution, with information regarding the use of the various variables / structures as well as the number of times they were accessed.

  Having these statistics enables us to better organize memory operations so as to not stall execution. It is also possible to find specific data structures that accommodate the needs of the program better.

- **Cache usage statistics**

  The aforementioned metrics are not tied to cache properties. Even so, it is still important to have a picture of the program's behavior, regarding the cache, in order to structure the code and organize the memory accesses in a way that it takes advantage of the memory hierarchy to improve the performance.

Another Valgrind tool, Cachegrind, is used for that purpose. Cachegrind provides us with all necessary information regarding the cache behavior of the program in question, such as number of references and hit and miss ratios.

Maximizing cache hits while minimizing misses,especially on the last level of the cache, is the desired behavior of the program's memory operations so as to achieve optimal memory performance.

- **Memory activity**

It is common , during program execution , to load / store data when they are needed / ready to be written. While this method seems logical it often hurts performance by stalling program execution for memory operations.

Using the order of memory accesses and the access pairs, stored in the Access list array, we are able to create a "timeline" of the accesses and display it on the activity graph. This visualization of the accesses can help us identify sporadic memory accesses that might harm performance.

Bursting, grouping up memory operations ,depending on the program, can help eliminate idle processor time waiting for operands from memory and even safeguard against data dependency issues especially when parallelizing code. It is therefore important to know the relative timing of the memory accesses during execution and their type so as to organize them better.

3.4 Visualization of results

So far we have captured all memory accesses in a trace file, filtered out all unnecessary data, processed them and organized them in useful structures. In this final stage of the implementation we visualize the results of the previous processing stages in a way that is meaningful and informative to the user and capable of leading him to useful conclusions. .

After researching the most common chart types capable of conveying the information we want to visualize , we decided to use a scatter plot to display reuse distances, a typical line plot for the average reuse distance and another scatter plot for the reuse distances standard deviation. The choice was based on the simplicity of these charts, being able to display information about the range of the reuse distances in the program as well as showing the trend that they follow with each added memory access. Bar graphs, another common and straightforward chart type, were chosen depict the number of accesses for individual variables / structures as well as the number of bytes accessed respectively. Finally, for the most important information we want to visualize, access patterns, we chose the event plot seeing as it is a very intuitive way to identify data locality , both spatial and temporal.

For the creation of the plots mentioned earlier we will use Matplotlib, a Python 2D plotting library that is used by thousands of scientists worldwide, is easy to use and

available across platforms [32].

3.5 User interface

While the initial design was for the tool to use a command-line interface (CLI) , simple and lightweight, it is obvious that the most important information the tool provides is portrayed with the use of graphs. In that regard , a new GUI was created in order to allow the user to handle the results easier whether they are in text or image format.

One of the most common graphical user interfaces (GUI) libraries and the de-facto standard for Python is Tkinter, an interface to the Tk GUI toolkit [33]. Given its wide use, it is included in the Python installation so no extra setup is required. It is intuitive to use, allows for multiple windows and widgets and also makes importing and handling graphs really easy.

Having decided on the tool to be used it was now time to design the tool's interface. The main focus of the design is for the interface to be simple and easy to use; after all “Simplicity is the ultimate sophistication” (Leonardo Da Vinci). An overview of the final graphical user interface can be seen in figure 14.

Figure. 14 Graphical user interface.

The GUI is separated in 3 main areas: the Menu bar, the Log window and the Button area. The **Menu bar** is a typical menu found in every program and contains the File and Help sub-menus providing functionalities such as opening a .C source file , retaining the output files from a previous analysis, using the files from a previous analysis, clearing the log window, saving the contents of the log window in a text file, exiting the program and

information about the tool and its license respectively.

The **Log window** is a scrollable frame where the results of each function of the tool are presented in full detail. During execution even the most simple programs can have thousands or millions of memory accesses. The amount of these accesses is not easily understood through a graph, plus the lowest values tend to disappear on the same graphs with disproportionately larger values. For example it is not easy to visualize small reuse distances (e.g 1-50) when you have reuse distances as big as 100.000 (sparsely accessed variables / structures). These details are accurately listed in the Log window.

The **Button area** consists of 4 self-explanatorily titled buttons, Reuse distance, Variable Metrics, Memory Activity Graph and Additional Information. The buttons are inactive when the tool is opened and become active after the program analysis is complete. The buttons code , upon activation, opens a new window, creates the required plots and displays them inside the window. The Memory Activity window offers a dropdown list where the user can select any individual variable / structure to see the memory accesses related to it, select consecutive values of the list to add them to the plot or select the value “All” to see the full plot of all the memory accesses of the program execution. A clear graph button is also provided for resetting the graph. Finally, the last button is Additional information. This button does not open a new window like the previous ones. It prints details about the tool's execution like execution time, created files sizes, trace file size reduction and cache statistics about the program in the log window.

A toolbar offering graph manipulation options is also embedded in each graph window. This toolbar allows the user to zoom in and pan the graphs, an extremely helpful functionality seeing how the graphs tend to be really dense, change the size of the graphs, return the graphs to their original forms and store the graphs as images.

While plain looking, the user interface is intuitive and easy to use. It manages to display all the necessary information in order to identify the program's memory access patterns and examine its locality, locate memory hotspots regarding variables / data structures and sparsely accessed data, collect profiling information regarding the program's execution time and lastly provide detailed cache statistics.

4

Evaluation

In this chapter we demonstrate the functionality of the tool and validate the produced outputs from using various source codes as input.

4.1 Matrix multiplication

The first source code used is matrix multiplication of two 10x10 matrices. Due to its few lines of code and small matrix size, the results are not too complicated and provide a simple example of how the tool works. In figure 15 we can see the visualization of the results regarding the reuse distance of data in our program.

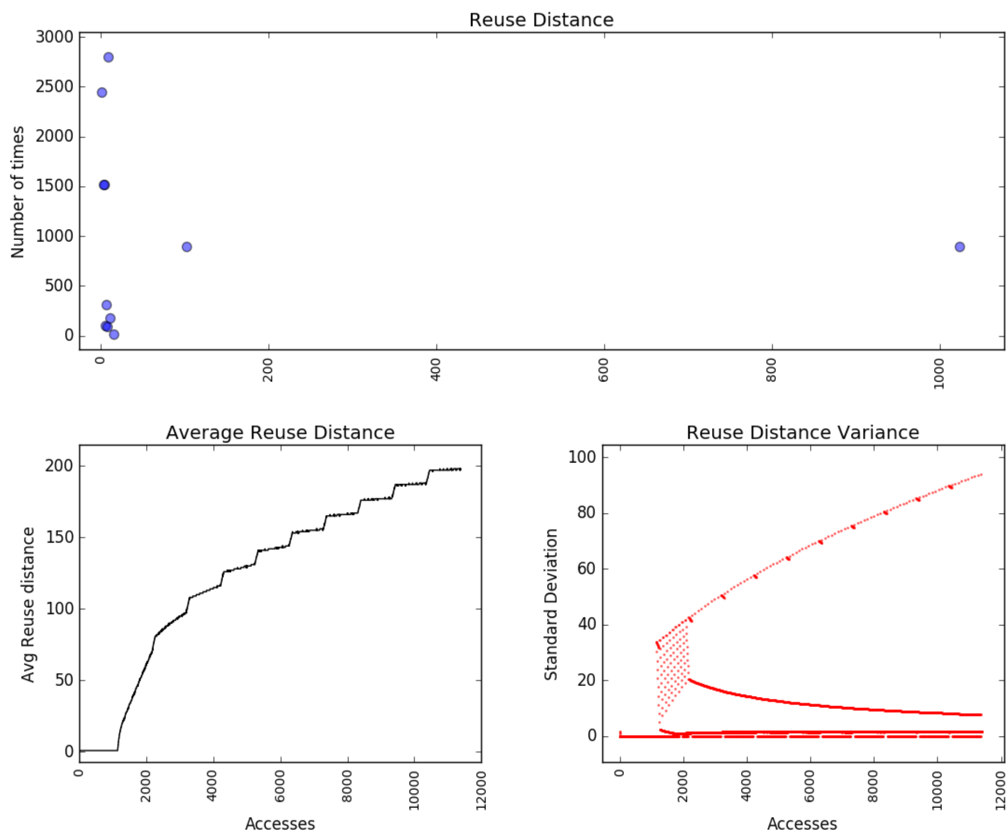

Figure. 15 Matrix multiplication reuse distance analysis graph.

In the top graph we plot the reuse distances of the program's data and the number of times for each one of them. The bottom two graphs show how the average reuse distance changes with each memory access and the standard deviation of individual reuse distance values. As we can see, reuse distance in this code is relatively small, something to be expected considering the small size of the matrices. The average reuse distance also has a predictable trend. Data accessed sequentially, leading to similar reuse distance between them, have a stabilizing effect on the average. This can be seen on the almost straight and parallel to the horizontal axis segments of the plot that correspond to the 10 values read sequentially from the second matrix. The standard deviation graph confirms our observations as well. Reuse distance values deviate from the average when data from a new row (from the first matrix) are accessed and are close to the average for the rest of the accesses. These graphs can help developers get a picture of the program's data locality and how the memory accesses are spread throughout the program's execution.

In figure 16 we can see the memory in bytes used and the number of accesses for the various variables / structures of the program.

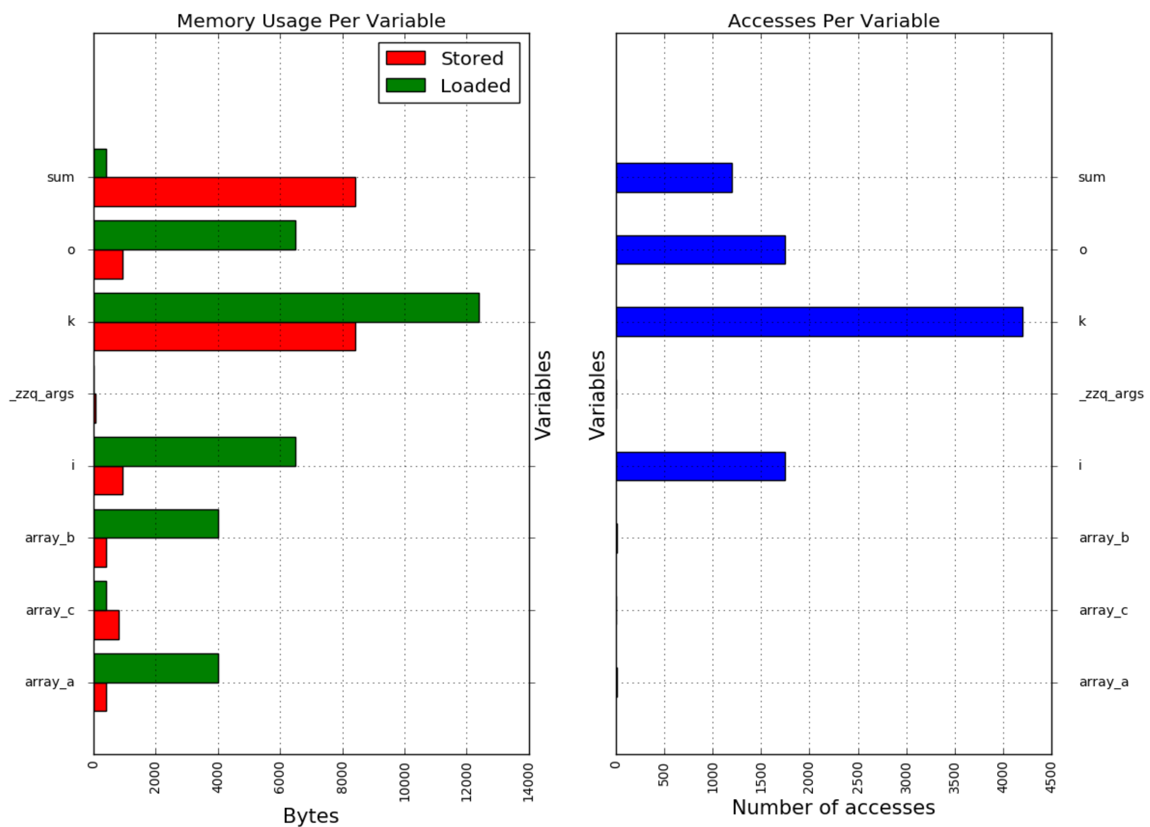

Figure. 16 Matrix multiplication variable metrics graph.

The most used variables, both in number of access and number of bytes stored / loaded, are the counters for accessing the values in the matrices. Therefore, using registers for these variables will have a positive impact on the program's performance, even more for larger matrices.

Having statistical information about the program being analyzed provides

opportunities for optimizations but is not indicative of the memory access patterns. The memory activity graph, seen in figure 17 is used for that purpose. On the X-axis we have the memory accesses in order spanning to the right and on the Y-axis we have the variables / structures used in the program.

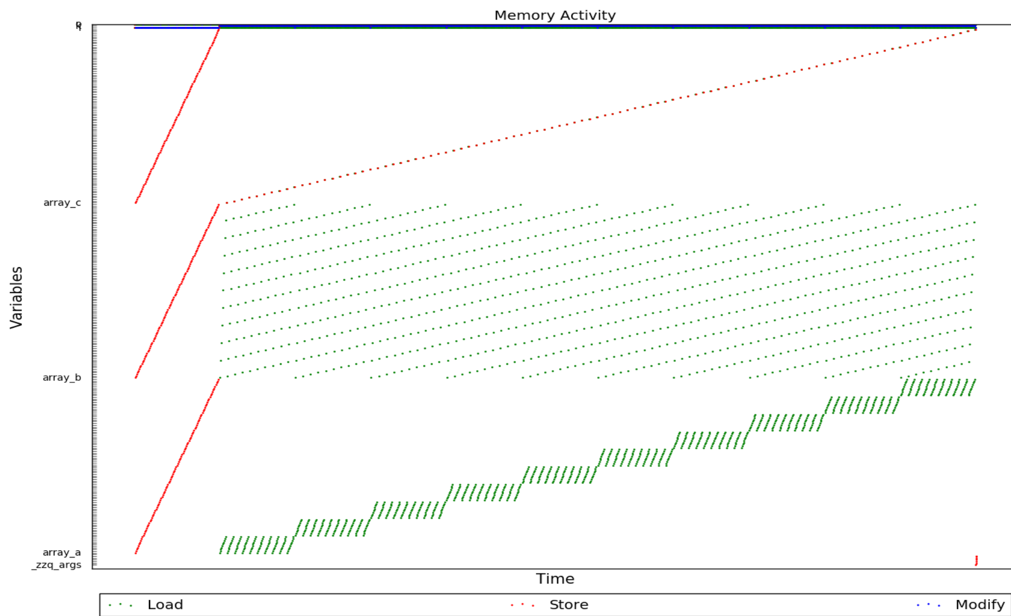

Figure. 17 Matrix multiplication memory activity graph.

The more memory accesses a program has the denser this graph will be along the horizontal axis. The number of data variables / structures controls the graph density on the vertical axis. The small matrix size works in our favor in this example, but for other cases with a lot more accesses and data elements the ability to zoom and pan in this graph is extremely helpful.

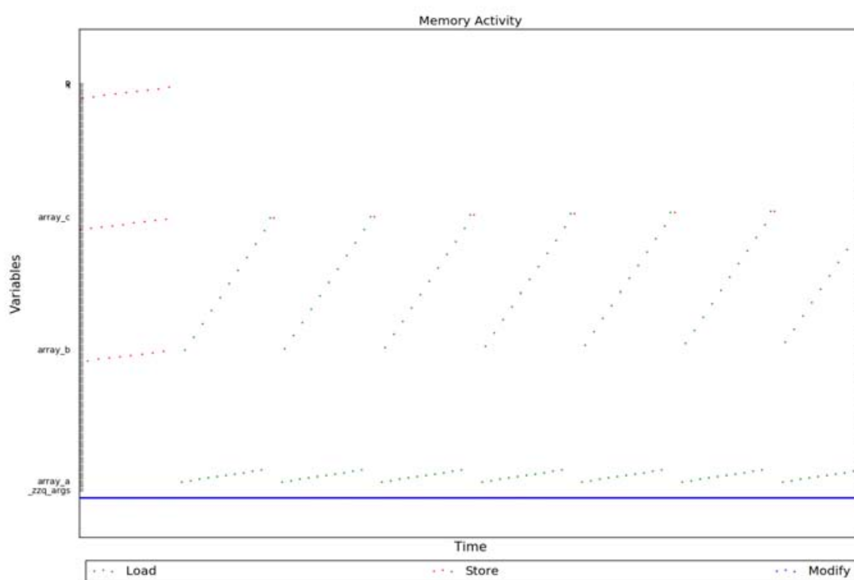

Figure. 18 Matrix multiplication memory activity graph (zoomed in).

Zooming in a part on the memory activity graph (fig. 18) provides us with more detail as to the accesses sequence and the time and spatial locality of the variables / structures used. Observing this graph, one can deduce, depending on his expertise, parts if not all of the code's functionality. In this example it is obvious that elements from rows in array_a are used for an operation with elements from columns in array_b and the result is stored in array_c. Visualizing the access patterns in this way provides an intuitive and informative method to identify patterns that may harm memory performance or unwanted accesses leading to false results.

Below we can see additional information regarding the tool's execution (fig. 19) generated during runtime. This information is useful as a mean to examine it's performance for various source codes.

```
Additional info:
---------------
Gleipnir memory trace filesize:  822.0 kB

Number of unique lut records:    310
lut filesize:                    37.5 kB
Memory accesses list filesize:   89.0 kB
Size reduced by:                 695.4 kB  ~ 84.61% of the original file size
Total memory accesses:           11391

Compiling source code time:                              0.163 s
Valgrind/Gleipnir execution time:                        12.413 s
Valgrind/Cachegrind execution time:                      0.278 s
Compiling gleipnir output manipulation code:             0.078 s
Gleipnir output manipulation code execution time:        0.032 s
Compiling data manipulation time:                        0.068 s
Data manipulation code execution time:                   0.037 s

Total execution time:                                    13.084 s
```

Figure. 19 Matrix multiplication execution statistics.

Encapsulating another Valgrind tool, Cachegrind, in our tool we are able to profile the program's cache behavior(fig. 20). This tool provides us with information about the number of instruction and data references in our program and their hits and misses respectively. The references are also separated in first (L1) and last (LL) level of the cache hierarchy. This information can be useful when trying to achieve cache-friendly behavior. Reducing the miss rate improves cache performance and subsequently memory performance of the program alltogether.

```
Cache statistics:
--------------------

I1 cache:          32768 B, 64 B, 8-way associative
D1 cache:          32768 B, 64 B, 8-way associative
LL cache:          6291456 B, 64 B, 12-way associative

I refs          180793
I1 misses       814
LLi misses      806
I1 miss rate    0.45%
LLi miss rate   0.45%

D refs          68281 ( 50431 rd + 17850 wr )
D1 misses       2110 ( 1498 rd + 612 wr )
LLd misses      1814 ( 1252 rd + 562 wr)
D1 miss rate    3.1% ( 3.0% + 3.4% )
LLd miss rate   2.7% ( 2.5% + 3.1% )

LL refs         2924 ( 2312 rd + 612 wr )
LL misses       2620 ( 2058 rd + 562 wr )
LL miss rate    1.1% ( 0.89% + 0.28% )
```

Figure. 20 Matrix multiplication cache statistics.

For comparison purposes, blocked execution of the same source code (matrix size is 10 and block size is 5) would produce the memory activity graph shown in figure 21.

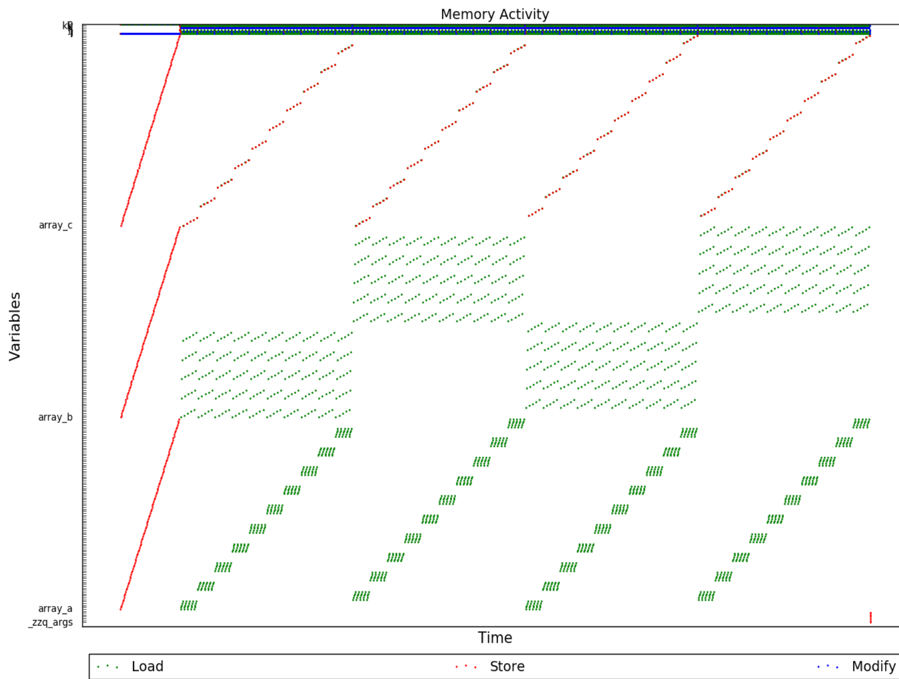

Figure. 21 Block Matrix multiplication memory activity graph.

This matrix size is too small for a noticeable difference in execution time. If we increase the matrix size to 100 and block size to 10 and compare the resulting graphs we can see how the blocked approach increases data locality. Comparing the average reuse distance of the two versions of the code, for instance, we can see a noticeable reduction (fig. 22).

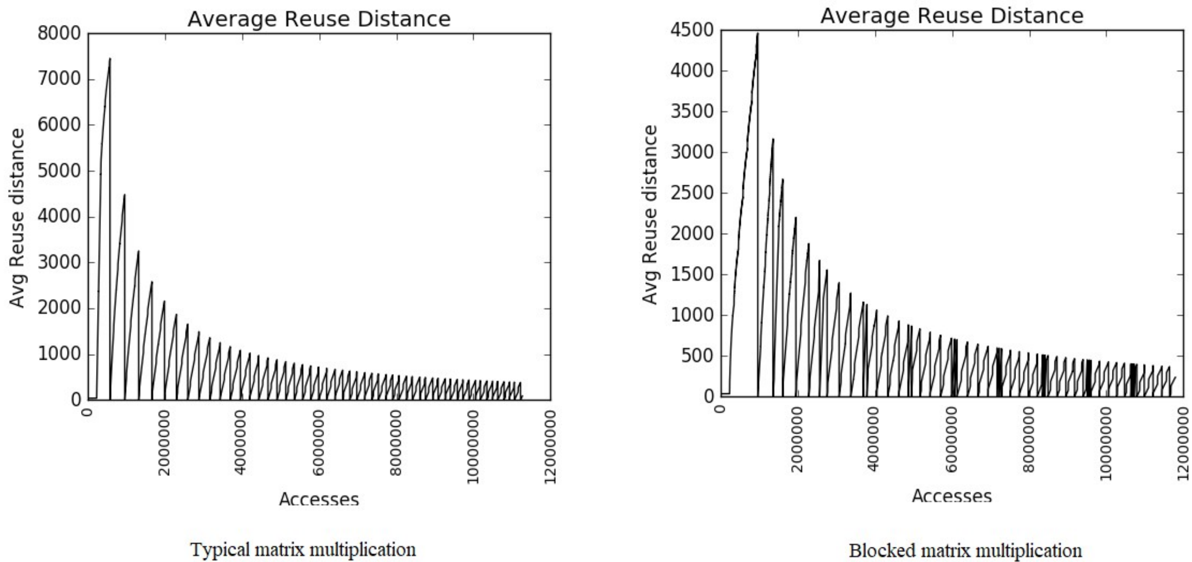

Figure. 22 Matrix multiplication average reuse distance comparison.

4.2 **Mandelbrot**

Programs are often not as small as matrix multiplication. Next we analyze a larger program that draws the Mandelbrot set in a .ppm image file. Starting with the reuse distance analysis, seen in figure 23, we can observe a small and stable average value. This happens, naturally, when the same set of variables are accessed with the same sequence throughout the program.

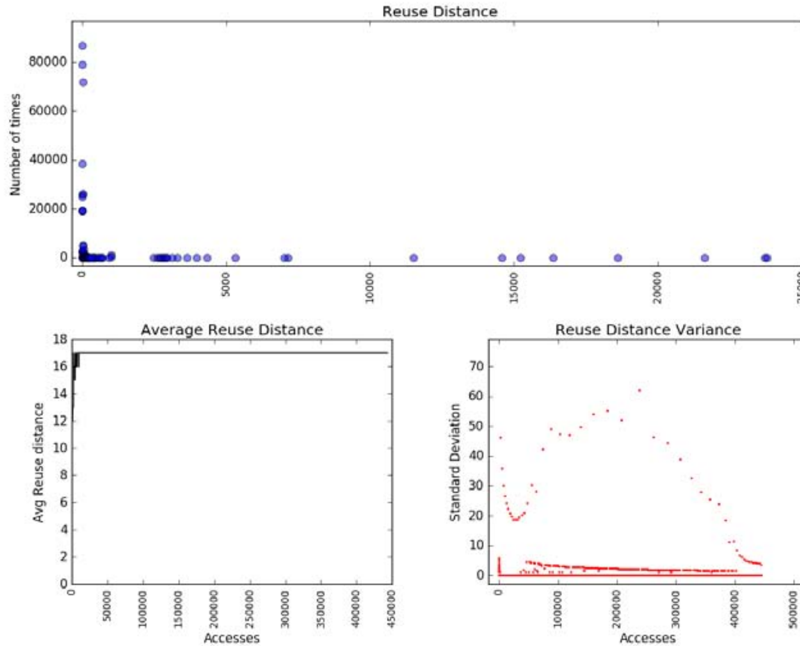

Figure. 23 Mandelbrot reuse distance analysis graph.

The variable use is shown in figure 24.

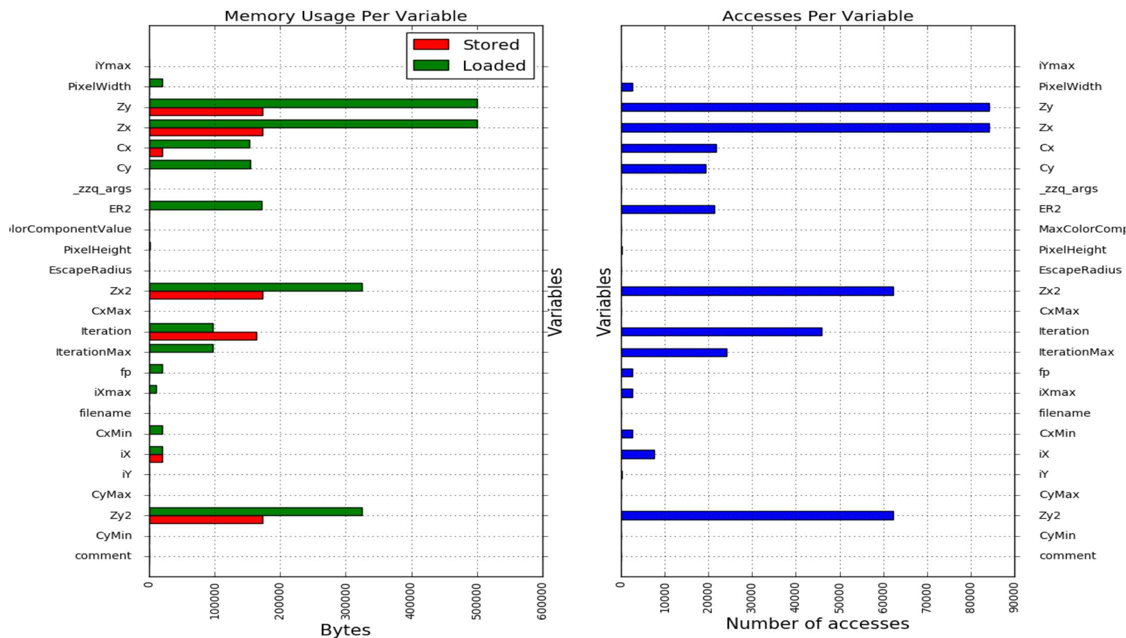

Figure. 24 Mandelbrot variable metrics graph.

It is obvious from the previous figure that the majority of the accesses occurs in a handful of variables used to calculate the set, an observation confirmed by the reuse distance analysis. From these two graphs we can make an educated guess about the patterns that will be visible in the memory activity graph(fig. 25). Seeing that only a few variables are accessed mostly we expect the graph to show clearly sequential access in these variables.

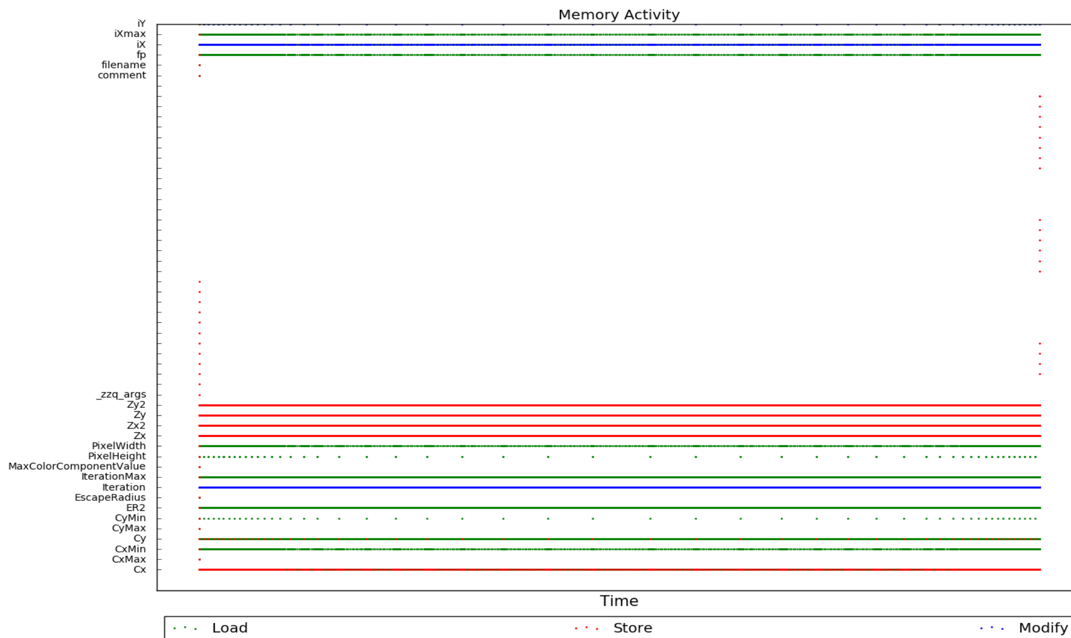

Figure. 25 Mandelbrot memory activity graph.

Due to the large amount of accesses (~445,000), the accesses to all variables appear to be straight lines. The proper amount of zooming in and panning can reveal the individual accesses and help discern their order, visible in figure 26.

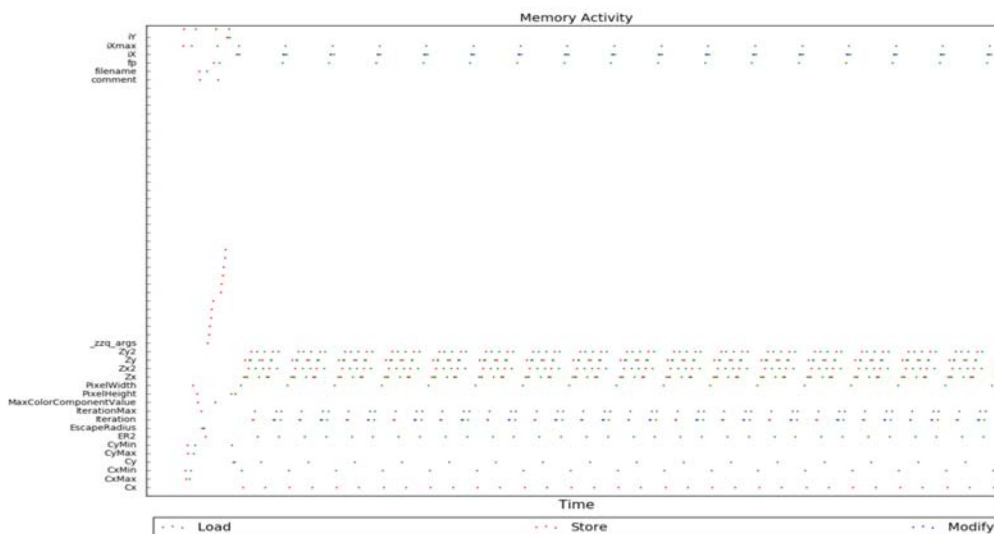

Figure. 26 Mandelbrot memory activity graph (zoomed in).

We can use figure 27 and figure 28 to compare the execution of this program with the previous one, mostly the scale in the number of accesses.

```
Additional info:
---------------
Gleipnir memory trace filesize:  22.52 MB

Number of unique lut records:    54
lut filesize:                    6.5 kB
Memory accesses list filesize:   3.38 MB
Size reduced by:                 19.13 MB  ~ 84.94% of the original file size
Total memory accesses:           443585

Compiling source code time:                              0.056 s
Valgrind/Gleipnir execution time:                        356.931 s
Valgrind/Cachegrind execution time:                      0.319 s
Compiling gleipnir output manipulation code:             0.086 s
Gleipnir output manipulation code execution time:        0.544 s
Compiling data manipulation time:                        0.073 s
Data manipulation code execution time:                   0.581 s

Total execution time:                                    358.610 s
```

Figure. 27 Mandelbrot execution statistics.

```
Cache statistics:
--------------------

I1 cache:         32768 B, 64 B, 8-way associative
D1 cache:         32768 B, 64 B, 8-way associative
LL cache:         6291456 B, 64 B, 12-way associative

I refs          1053112
I1 misses       1018
LLi misses      1002
I1 miss rate    0.097%
LLi miss rate   0.095%

D refs          640640 ( 481756 rd + 158884 wr )
D1 misses       3040 ( 2414 rd + 626 wr )
LLd misses      2500 ( 1940 rd + 560 wr)
D1 miss rate    0.47% ( 0.5% + 0.39% )
LLd miss rate   0.39% ( 0.4% + 0.35% )

LL refs         4058 ( 3432 rd + 626 wr )
LL misses       3502 ( 2942 rd + 560 wr )
LL miss rate    0.21% ( 0.19% + 0.046% )
```

Figure. 28 Mandelbrot cache statistics.

4.3 **Bzip2**

Accessing the same data repeatedly, as seen with the Mandelbrot program, or accessing array data sequentially, like in matrix multiplication, produce straightforward access patterns. Other programs, however, have more irregular and unpredictable memory access patterns. One such example is the bzip2 program which is used for data compression. Bzip2 uses the Burrows-Wheeler block sorting text compression algorithm and Huffman coding, and compresses data in blocks, so block patterns of reading and writing are expected on the graph. Due to the nature of this program we are going to compare side by side the respective graphs for compression and decompression.

An educated guess about the program's reuse distance can be done based on that when decompressing the memory accessed has good spatial locality thus lowering the average. Reuse distances, for decompressing data with the Huffman tree, should have

higher variance due to the randomness of the set of symbols being decompressed. In figure 29 and figure 30 we can see the graphs of the reuse distance analysis when compressing and decompressing.

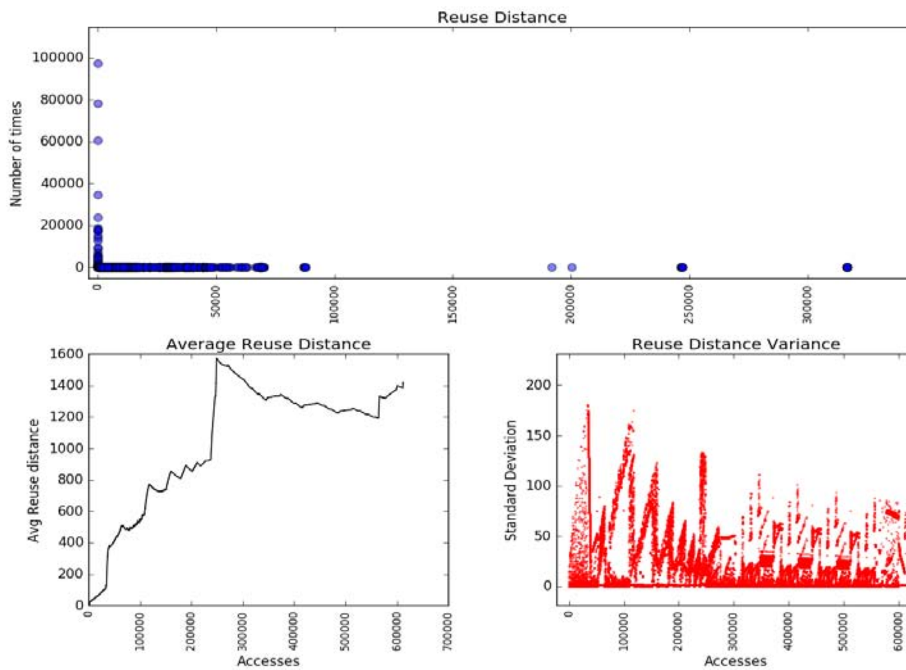

Figure. 29 Bzip2 (compression) reuse distance analysis graph.

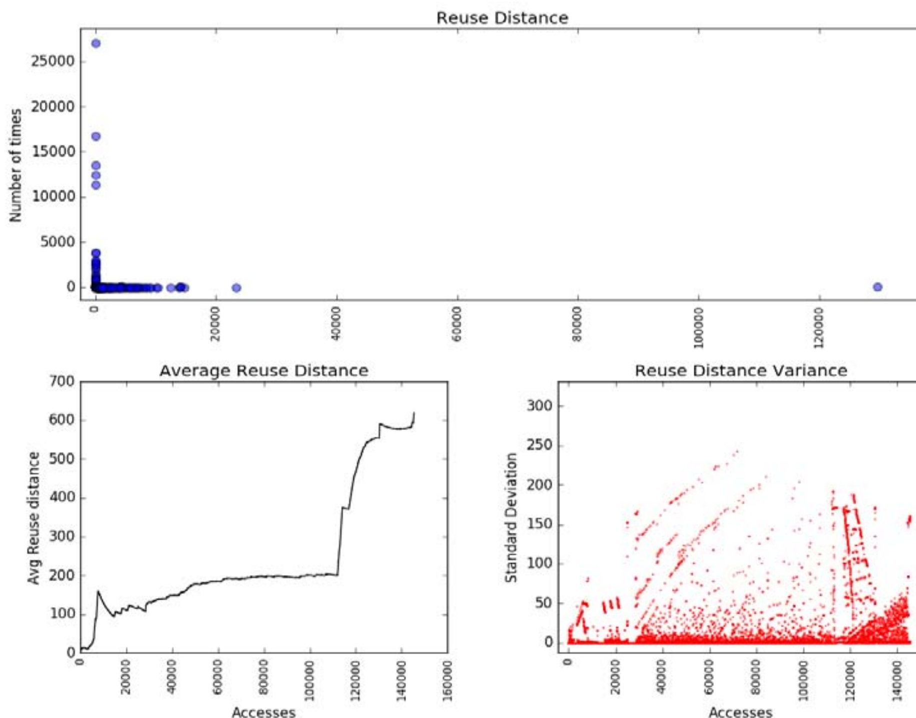

Figure. 30 Bzip2 (decompression) reuse distance analysis graph.

The results confirm our guesses. The Variable metrics graphs, even though they differ because of the different functionalitiy the two executions implement, can be seen in figure 31 and figure 32 as a reference.

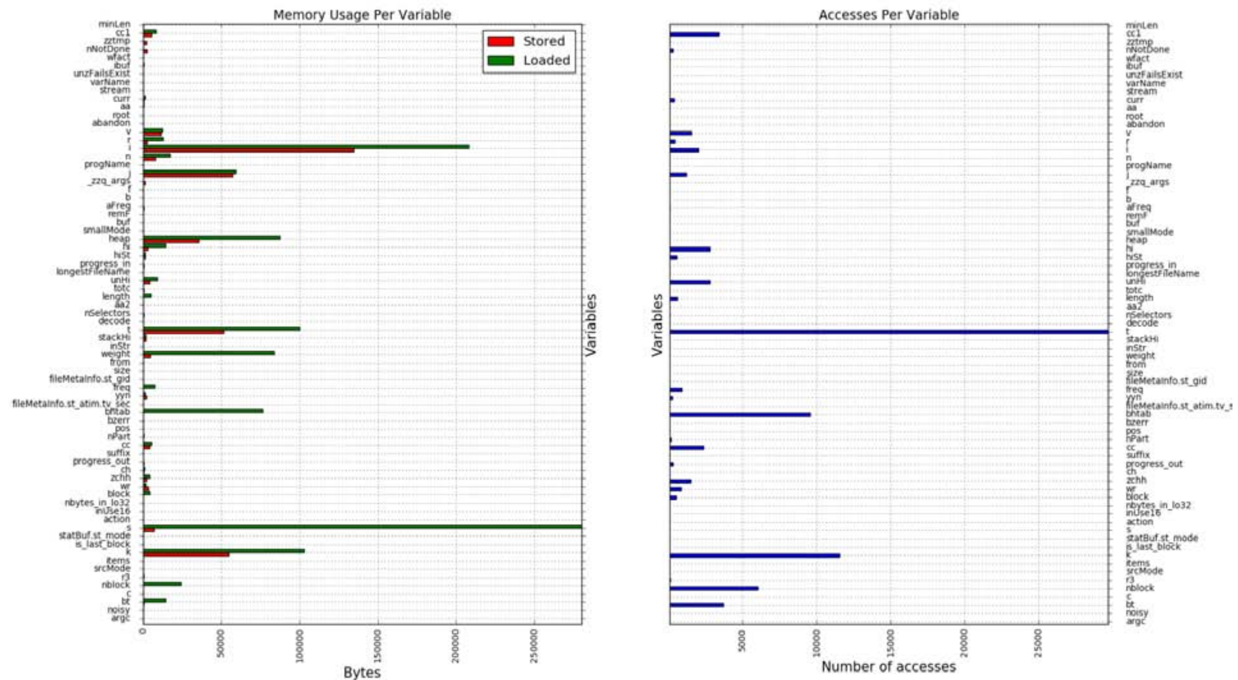

Figure. 31 Bzip2 (compression) variable metrics graph.

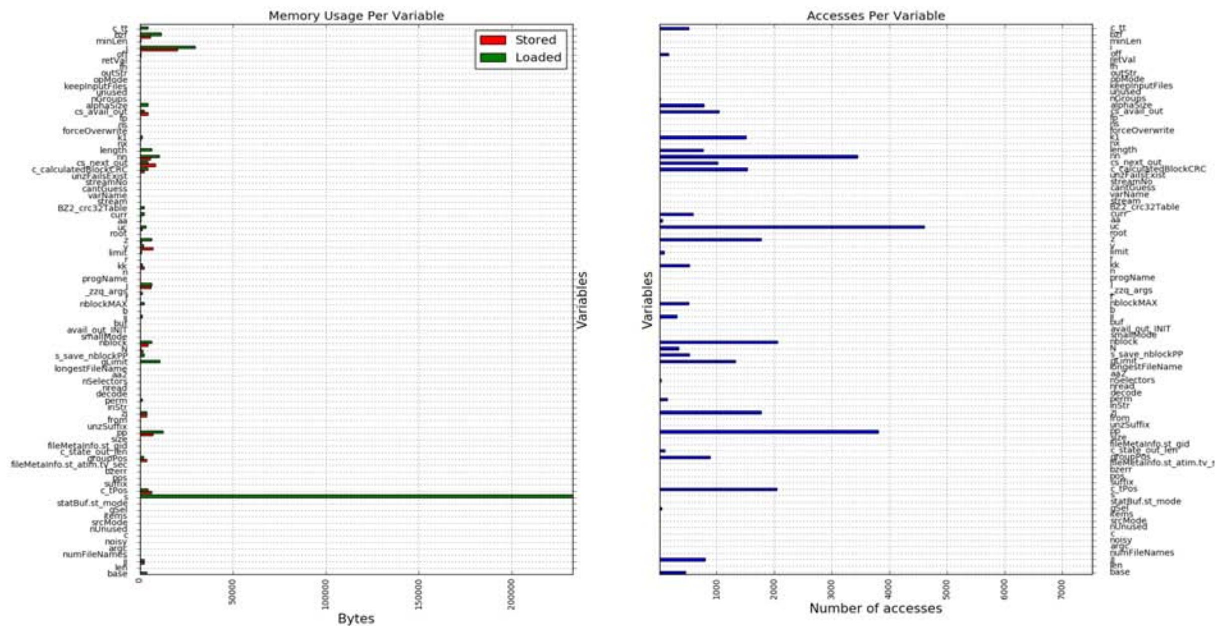

Figure. 32 Bzip2 (decompression) variable metrics graph.

The memory activity graph is expected to be the most distinctly different since the patterns with which we access data and the different stages of execution (reading from input, processing data, writing to output) are different. In figure 33 and figure 34 we can see the graphs in question.

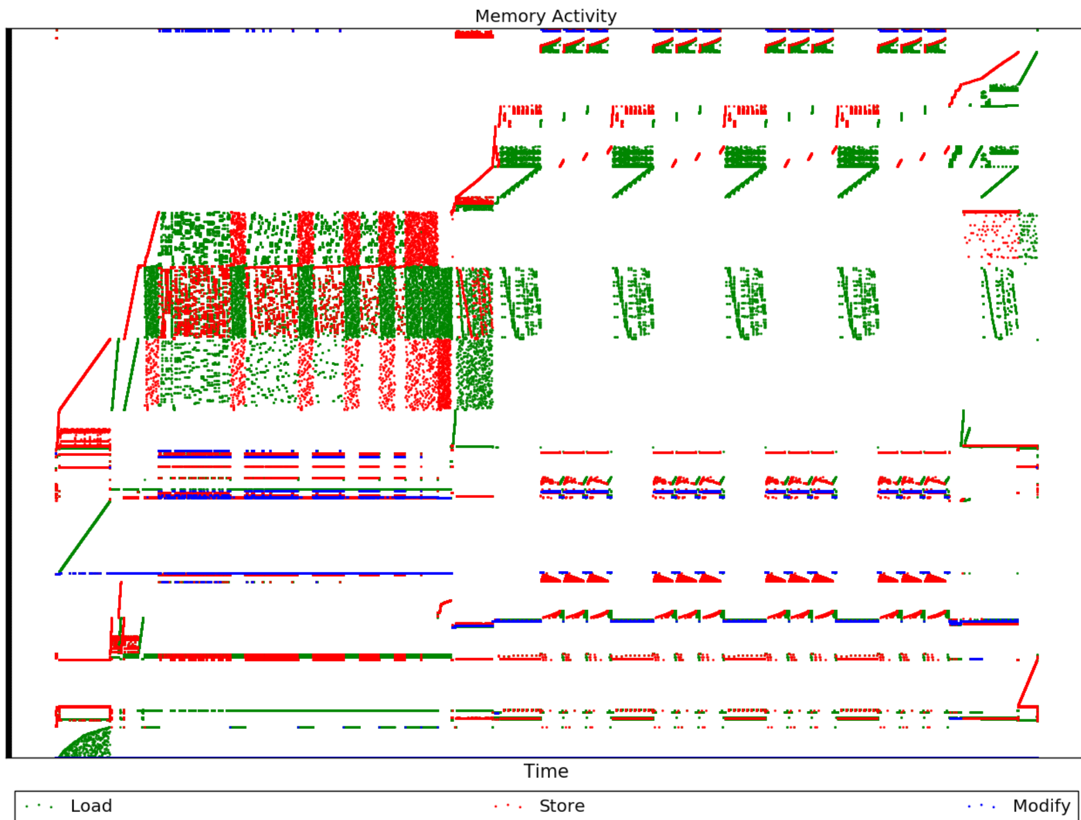

Figure. 33 Bzip2 (compression) memory activity graph.

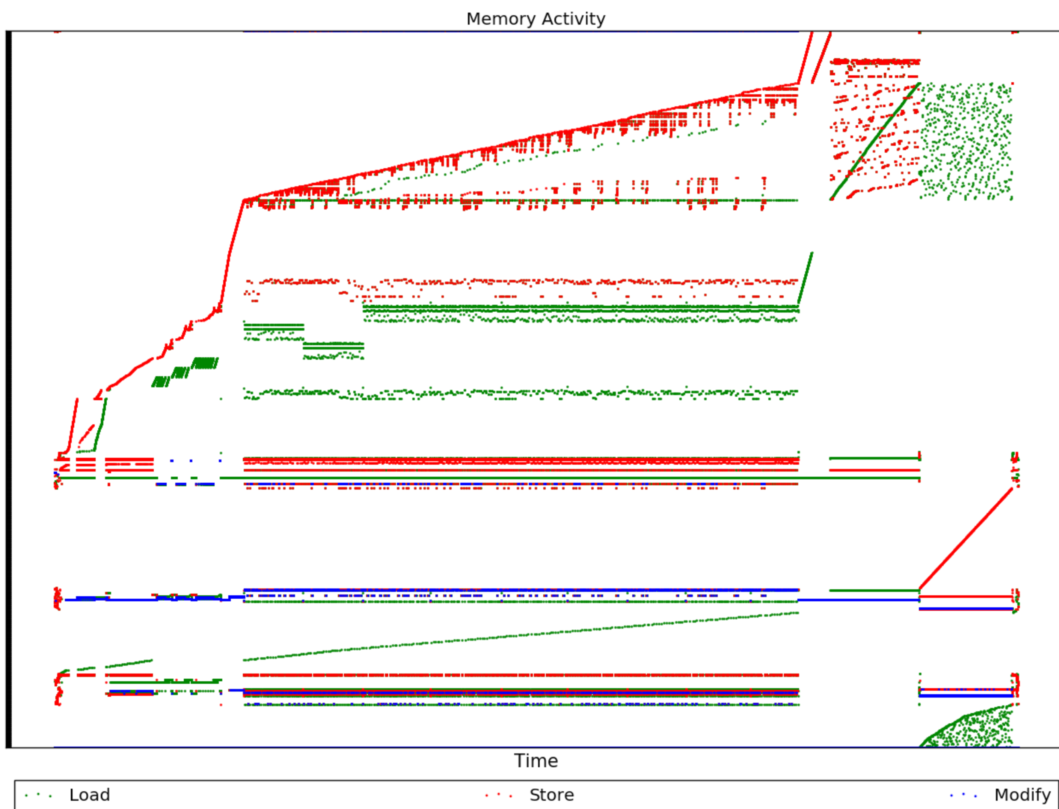

Figure. 34 Bzip2 (decompression) memory activity graph.

In the above graphs, one can easily visually identify the different stages (input,processing,output) of execution taking place during runtime of the respective programs.

The memory access patterns of the program's execution is pictured in full detail on the memory graph. Observations about the program's usage of data structures, data locality,most time consuming operations and general stages of the program's executions can be easily made from this graph. In the hands of software developers and researchers it is a valuable tool offering insight into the inner workings of the program and its memory behavior.

5

Related Work

The interaction to the memory subsystem is one of the main targets of performance optimization of programs. The past few years there has been a significant increase in the creation of specialized tools capable of profiling memory in great detail, recording program memory traces , supporting singlethreaded and multithreaded applications,single, multi and manycore processors, UMA and NUMA architectures. Despite the variety in focus areas,capabilities, platform availability and licenses of the available tools, most of them are based on one of three dynamic binary instrumentation (DBI) frameworks, Pin, Valgrind and DynamoRIO.

Pin is a DBI framework developed by Intel that enables the creation of dynamic program analysis tools [34]. The tools created with Pin are called Pintools. Perhaps the most notable pintool used in memory profiling, debugging and performance analysis is Intel's VTune. VTune is a commercial application for software performance analysis heavily used in the HPC industry because of its support for multiple programming languages, operating systems, architectures and a plethora of features such as software sampling, hardware event sampling, memory access analysis and storage analysis. QUAD [36] is another pintool proposed by researchers that supports multiple platforms and operating systems. It has been designed as a system whose main goal is to identify data-dependencies at function level and identify opportunities for optimizations. One other notable pintool is TABARNAC [37], focused on NUMA architectures, data and thread mapping and the distribution of accesses by threads and structures.

Valgrind is an open-source DBI framework actively developed and maintained by multiple developers worldwide and licensed under the GNU license. Notable tools, similar to Gleipnir, created around this framework focusing on memory access profiling are Datagrind, mmtrace ,memhist and memview. All of them are made from independent developers with the goal of capturing the memory trace of a program and providing some basic statistics for the accesses. A notable distinction is that memview offers a visualization of the trace file as a 2D grid map of the memory.

DynamoRIO is an open-source DBI framework created as a collaboration between MIT and Hewlett-Packard. Commonly used tools built on DynamoRIO are Dr.Memory [38], TaintTrace [39] and Adept [40]. Dr.Memory is a memory monitoring tool that supports a multitude of platforms and operating systems. It is capable of capturing memory accesses and identifying memory-related programming errors. TaintTrace examines a program's dynamic behavior and keeps track of the propagation of untrusted (tainted) data during

program execution. It is mostly used in the security field and tainted data may represent sources such as user input, packets from the network, or data read from memory, specific files or devices. Finally, Adept is a dynamic execution profiling tool able to handle multi-threaded code by maintaining runtime profile information for different threads simultaneously. It can also profile events for dynamically generated code and self-modified code.

6

Conclusion

Today advances are made at an incredible rate in all fields of computing. Even so, and especially in high-performance computing, memory performance is the major performance bottleneck. It is,thus, imperative for those active mostly in the HPC field and secondarily in other performance-sensitive domains to analyze and consdier the memory properties of the system and program they are working on. To that purpose, knowledge of the memory properties, like hierarchy, and the memory behavior of the program in question are necessary. While the memory properties of a system are constant and known beforehand, every program's memory behavior differs and even more it changes with modifications made to the code or even with different input data. A means to observe and analyze that behavior is needed.

In this thesis we developed a tool for extracting memory accesses from trace files created using Gleipnir, a tool for the DBI framework Valgrind. Furthermore, after analyzing the extracted data we are able to identify the program's memory access patterns and visualize them in meaningful ways using graphs and statistics. The main goal was, after identifying them, to discern memory hotspots and their cause and provide assistance to programmers and researchers with finding ways to optimize memory performance of the program in question.

Use of the tool, as seen in chapter 4, provides intuitive and useful results about the program's access patterns and memory use. Information about the program's locality, easily noticeable on the memory activity graph combined with the order of accesses, are also crucial in identifying hotspots and data dependencies both of which are detrimental to the performance especially in multithreaded programs. A simple method to discover optimizations is using the tool to observe the performance impact and memory behavior of grouping memory operations. Discovering sparsely accessed data and hotspots provides necessary information in order to write cache-friendly code, maximizing data locality and ultimately optimizing memory performance.

6.1 Future work

The code needs to be optimized for more efficient memory usage and execution speed, mostly because of Gleipnir. According to its developers these are known issues and work is being done to reduce its memory use and overhead. In a similar manner the

other functionalities of our tool use a significant amount of memory, handling all the data extracted from the trace file. Rewriting parts of the code in Python ,due to a wide variety of libraries for handling data in more efficient structures will help reduce the tool's memory footprint. Adding support for multithreading will provide a significant speedup in execution due to the nature of the most time consuming tasks of the tool , which are parsing input from independent buffers and calculations. Furthermore,giving developers the ability to focus on a specific range of addresses or specific data structures beforehand will allow for more precise and detailed observations. Other parts that could use an improvement as well are the GUI, by adding more options and a command-line interface (CLI), and the cache analytics, by adding more detailed cache statistics and cache behavior visualization.

Bibliography

[1] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. 2013. Instant loading for main memory databases. Proc. VLDB Endow. 6, 14 (September 2013), 1702-1713. DOI=http://dx.doi.org/10.14778/2556549.2556555.

[2] D. A. Patterson and J. L. Hennessy , "Computer Architecture: A quantitative approach", 2nd edition. San Francisco, CA: Morgan Kaufmann Publishers,1996.

[3] Sally A. McKee. 2004. Reflections on the memory wall. In Proceedings of the 1st conference on Computing frontiers (CF '04). ACM, New York, NY, USA, 162-. DOI=http://dx.doi.org/10.1145/977091.977115 .

[4] John D. McCalpin,(2004) , "Perspective on the "Memory Wall" " [Online]. Available: https://www.cs.virginia.edu/~mccalpin/MemoryWall2004-02-20.ppt.

[5] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: implications of the obvious. SIGARCH Comput. Archit. News 23, 1 (March 1995), 20-24. DOI=http://dx.doi.org/10.1145/216585.216588.

[6] David Collin (2010) , "Data oriented design" [Online]. Available: http://www.dice.se/wp-content/uploads/2014/12/Introduction_to_Data-Oriented_Design.pdf [Accessed: 25- May-2017].

[7] B. Jang, D. Schaa, P. Mistry and D. Kaeli, "Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 105-118, Jan. 2011. doi: 10.1109/TPDS.2010.107 .

[8] Siddhartha Jana, Joseph Schuchart, and Barbara Chapman. 2014. Analysis of Energy and Performance of PGAS-based Data Access Patterns. In Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models (PGAS '14). ACM, New York, NY, USA, , Article 15 , 10 pages. DOI=http://dx.doi.org/10.1145/2676870.2676882.

[9] Nakano Y., Cid C., Kiyomoto S., Miyake Y. (2013) Memory Access Pattern Protection for Resource-Constrained Devices. In: Mangard S. (eds) Smart Card Research and Advanced Applications. CARDIS 2012. Lecture Notes in Computer Science, vol 7771. Springer, Berlin, Heidelberg.

[10] Anne Canteaut, Cédric Lauradoux, André Seznec. Understanding cache attacks. [Research Report] RR-5881, INRIA. 2006. <inria-00071387>.

[11] Toy, Wing; Zee, Benjamin (1986). Computer Hardware/Software Architecture. Prentice Hall. p. 30. ISBN 0-13-163502-6.

[12] R. Bryant and D. O'Hallaron, Computer systems, 3rd ed. 2001, p. 295.

[13] Weidendorfer, "Analysis and Optimization of the Memory Access Behavior of Applications", Université de Strasbourg, 2014.

[14] Kowarschik M, Weiß C (2003) An overview of cache optimization techniques and cache-aware numerical algorithms. In: Meyer U, Sanders P, Sibeyn J (eds) Algorithms for memory hierarchies: advanced lectures, vol 2625. Springer, Berlin, pp 213–232.

[15] Rogue Wave Software, "CPU Cache Optimization: Does It Matter? Should I Worry? Why?" [Online]. Available: https://www.roguewave.com/getattachment/b6524fa0-2f6f-4498-9875-194886ca8def/CPU-Cache-Optimization?sitename=RogueWave.

[16] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. 1991. The cache performance and optimizations of blocked algorithms. SIGOPS Oper. Syst. Rev. 25, Special Issue (April 1991), 63-74. DOI=http://dx.doi.org/10.1145/106974.106981.

[17] Naylor M., Runciman C. (2008) The Reduceron: Widening the von Neumann Bottleneck for Graph Reduction Using an FPGA. In: Chitil O., Horváth Z., Zsók V. (eds) Implementation and Application of Functional Languages. IFL 2007. Lecture Notes in Computer Science, vol 5083. Springer, Berlin, Heidelberg.

[18] Chuck Paridon. "Storage Performance Benchmarking Guidelines - Part I: Workload Design"[Online].Available:http://www.snia.org/sites/default/files/PerformanceBenchmarking.Nov2010.pdf [Accessed: 27- May- 2017].

[19] Rogue Wave Software, "Acumem ThreadSpotter" [Online]. Available: http://docs.roguewave.com/threadspotter/2011.1/manual/index.html [Accessed: 27- May- 2017].

[20] Wen-mei W. Hwu.'Algorithms,Implementations, and Evaluations', in GPU Computing Gems Emerald Edition (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011, p. 51-54.

[21] M. Thompson, "Memory Access Patterns Are Important", Mechanical-sympathy.blogspot.gr, 2017. [Online]. Available: https://mechanical-sympathy.blogspot.gr/2012/08/memory-access-patterns-are-important.html [Accessed: 27- May- 2017].

[22] Demaine, E.D.: Cache-oblivious data structures and algorithms. In: Proc. EFF summer school on massive data sets. LNCS, Springer, Heidelberg (2004).

[23] H. Prokop, 'Cache-Oblivious Algorithms', MSc, Massachusetts Institute of Technology, 1999.

[24] M. F. Sakr, Steven P. Levitan, Donald M. Chiarulli, Bill G. Horne, and C. Lee Giles. 1997. Predicting Multiprocessor Memory Access Patterns with Learning Models. In Proceedings of the Fourteenth International Conference on Machine Learning (ICML '97), Douglas H. Fisher (Ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 305-

312.

[25] Yasuo Ishii, Mary Inaba, and Kei Hiraki. 2009. Access map pattern matching for data cache prefetch. In Proceedings of the 23rd international conference on Supercomputing (ICS '09). ACM, New York, NY, USA, 499-500. DOI=http://dx.doi.org/10.1145/1542275.1542349.

[26] Akanksha Jain and Calvin Lin. 2013. Linearizing irregular memory accesses for improved correlated prefetching. In Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46). ACM, New York, NY, USA, 247-259. DOI=http://dx.doi.org/10.1145/2540708.2540730.

[27] Tomislav Janjusic and Krishna Kavi. 2013. Gleipnir: a memory profiling and tracing tool. SIGARCH Comput. Archit. News 41, 4 (December 2013), 8-12. DOI=http://dx.doi.org/10.1145/2560488.2560491.

[28] N. Nethercote, J. Seward, Valgrind: a framework for heavyweight dynamic binary instrumentation, in: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, PLDI ’07, ACM, New York, NY, USA, 2007, pp. 89-100. DOI:http://doi.acm.org/10.1145/1250734.1250746.

[29] T. Janjusic, C. Kartsaklis, and W. Dali. Scalability analysis of gleipnir: A memory tracing and profiling tool, on titan. In Cray User Group, May 2014.

[30] Yutao Zhong, Xipeng Shen, and Chen Ding. 2009. Program locality analysis using reuse distance. ACM Trans. Program. Lang. Syst. 31, 6, Article 20 (August 2009), 39 pages. DOI=http://dx.doi.org/10.1145/1552309.1552310.

[31] Chen Ding and Zhong, Y. 2001. Reuse Distance Analysis. Technical Report. University of Rochester, Rochester, NY, USA.

[32] John D. Hunter. 2007. Matplotlib: A 2D Graphics Environment. Computing in Science and Engg. 9, 3 (May 2007), 90-95. DOI: https://doi.org/10.1109/MCSE.2007.55.

[33] Shipman, John W. (2010-12-12), Tkinter reference: a GUI for Python,[Online]. Available: http://infohost.nmt.edu/tcc/help/pubs/tkinter/ [Accessed: 12- June- 2017]

[34] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. SIGPLAN Not. 40, 6 (June 2005), 190-200. DOI=http://dx.doi.org/10.1145/1064978.1065034.

[35] Corina,M. "Quantitative analysis and visualization of memory access patterns" TU Delft, Delft University of Technology,2010.

[36] Ostadzadeh S.A., Meeuws R.J., Galuzzi C., Bertels K. (2010) QUAD – A Memory Access Pattern Analyser. In: Sirisuk P., Morgan F., El-Ghazawi T., Amano H. (eds) Reconfigurable Computing: Architectures, Tools and Applications. ARC 2010. Lecture Notes in Computer Science, vol 5992. Springer, Berlin, Heidelberg.

[37] David Beniamine, Matthias Diener, Guillaume Huard, and Philippe O. A. Navaux. 2015. TABARNAC: visualizing and resolving memory access issues on NUMA architectures. In Proceedings of the 2nd Workshop on Visual Performance Analysis (VPA '15). ACM,New York,NY,USA, Article 1,9 pages. DOI: https://doi.org/10.1145/2835238.2835239.

[38] Derek Bruening and Qin Zhao. 2011. Practical memory checking with Dr. Memory. In Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11). IEEE Computer Society, Washington, DC, USA, 213-223.

[39] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. 2006. TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In Proceedings of the 11th IEEE Symposium on Computers and Communications (ISCC '06). IEEE Computer Society, Washington, DC, USA, 749-754. DOI=http://dx.doi.org/10.1109/ISCC.2006.158.

[40] Qin Zhao, Joon Edward Sim, Weng-Fai Wong, and Larry Rudolph. 2006. DEP: detailed execution profile. In Proceedings of the 15th international conference on Parallel architectures and compilation techniques (PACT '06). ACM, New York, NY, USA, 154-163. DOI=http://dx.doi.org/10.1145/1152154.1152180.