MVIZION: TAKING A CLOSER LOOK AT MEMORY UTILIZATION IN C/C++ PROGRAMS

BY

NADIA NIKOLAYEVNA TKACH

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Adviser:

Professor Roy H. Campbell

Abstract

Computer system performance depends on the efficiency and precision of all of its components. When it comes to software, the primary goal is to ensure the most effective use of the hardware resources while maintaining the accuracy of the computations being handled. While many software developers focus their attention on the running time complexity and parallelism of computations in terms of CPU utilization; there is an evident lack of memory map analysis and RAM usage efficiency. mVizion is a novel visualization tool, combined with a digital forensic tool Cafegrind, shows how different memory utilization patterns are being used by different C/C++ programs, running in Unix/Linux environments. Cafegrind is used to produce different traces of heap memory accesses, including both allocations and deallocation, from different applications. mVizion is then used to interpret these traces, and through the use of multiple windows present detailed information about each distinct data structure; thus offering a comprehensive view and deeper understanding of a program inner memory workings. The evaluation of mVizion shows a fair performance and its capacity to scale to large log data files.

To my

Mom

Acknowledgments

I'd like to thank my advisor, professor Roy Campbell, and my mentor, Alejandro Gutierrez, for the support, leadership, motivation, and invaluable input during my graduate studies at Illinois. I greatly appreciate the help and input on the project from Shivaram Venkataraman and Ellick Chan, and I sincerely enjoyed working with them. Also I'd like to extend my gratitude to Rhonda McElroy, Laura Baylor, Cinda Heeren, and all members of Department of Computer Science academic office for their support and direction during the tough times. I'd also like to thank the Systems Research Group I've been a part of my last year at Illinois and all my professors and friends in Computer Science for all the help during my undergraduate and graduate studies and for providing a warm feeling of welcomeness and inclusiveness. And last, but not least, I would like to thank my Mom for always being there for me, providing support and motivation, and helping me make clearheaded decisions throughout all the years.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Thesis Statement: Visualization helps analyze memory performance and understand memory behavior in order to improve memory utilization in future applications.

As software applications grow more and more complex in response to users needs and expectations; the software products require a better quality assurance, precision and efficiency in utilization of users' resources, both hardware and software. In order to improve an application's performance, one must view and understand the complete systems architecture and work at each of the individual layers in order to improve the utilization and performance of all its components. This includes a thorough analysis of CPU usage, cache, as well as RAM utilization among others resources. While there have been previous efforts in profiling and analyzing these resources, an understanding of memory usage and performance has been to the best of my knowledge rather limited.

The lack of visualization tools to have a full comprehensive view of heap utilization makes understanding the memory usage and program behaviour a difficult task to accomplish. What automated scripts, trace dumps, and memory snapshots cannot achieve is to identify the patterns that govern memory management. Statistical analysis of such logs, or unit testing of software applications, along with underline libraries and dependencies are not able to provide a complete view of a systems behavior and the inner workings of the software that it is running.

A new trend in software system analysis is the ability to track, understand, and pinpoint unusual memory behavior during the software run-time. mVizion is a visualization tool designed to aid in the analysis of C/C++ program trace logs collected using a forensic profiling tool called Cafegrind. mVizion presents a heap memory map and its usage through time. It maintains a sequential list of all data structures created throughout the application's run-time; and tracks their type, size, memory address, allocation/de-allocation time, access operations (including both reads and writes accesses), as well as function call traces. Using multiple windows mVizion is able to present all of this information and provide software developers with a detailed view of how memory management is being handled at lower levels of the application.

mVizion uses the logs provided by Cafegrind to present detailed information about all allocated data structures during the run-time of each application; thus offering a snapshot view of the memory state at any point in time. Furthermore this timeline includes: specific information about the time when objects are allocated/de-allocated; number of block re-allocations and their new memory spaces; as well the current number of loads/stores of each data structure, and their respective addresses. Additionally mVizion infers information about dirty memory address spaces; non-de-allocated blocks at the end of application runtime; and the memory addresses accessed the most and the least within each objects memory space span.

mVizion presents a novel way of visualizing the heap memory map, that software developers can use to identify memory leaks and residue objects remaining in memory. Through the use of multiple windows mVizion can show in a visual manner detailed information about memory behaviour patterns of many C/C++ programs on Unix/Linux. This information can be used by software developers when designing or improving garbage collectors and memory allocators as well as redefining the policies that govern the memory. Likewise mVizion can help track and analyze memory bugs, detect memory leaks, monitor re-allocation patterns, churn, coalescence, memory fragmentation, memory space usage and locality.

Chapter 2

Background

2.1 System visualizations

There exists a number of related works in the area of memory map analysis and visualization offering a variety of features and functionalities when compared with mVizion. Among the many differences between mVizion and other work are the programming language, the platform requirements, and the operations procedures. The following includes a description of the contributions, limitations and differences of several projects when compared to mVizion. They are grouped into the following categories: heap visualization of different applications in Java versus C/C++; different procedural approaches; as well as static versus dynamic analysis tools.

2.2 Parallel, generic and special case visualization

Parallel programming and multi-threaded applications have been a long lasting research topic over the years. There exists a number of open source and commercial solutions for tracking and visualizing systems performance and operations. For instance, DRAGON [6] is an interactive analysis tool for optimization and parallelization of large programs that provides a visualization of call graphs, flow graphs, and data dependencies. Jive [12] and Jove [15] are another set of visualization tools developed for Java programs for gathering and presenting information about different threads belonging to an application. They include the threads state, the threads blocks and dependencies, flow analysis, function and synchronization calls, as well as the threads allocations/de-allocations, time and places of execution. These tools have been developed primarily for tracking and analyzing multi-threaded applications and don't provide a full view of memory utilization.

A tool for analyzing security features of software application has been proposed in TaintBochs [8]. While it lacks a visual component, it is able to log all the accesses to tainted data objects used by an application at the hardware level. TaintBochs tracks data objects states, tracks application components having access, writing to, or copying the data objects, and collects the time stamps for all the operations. [17] is a profiling, pattern discovery and performance analysis tool for cache memories optimization. It is capable of collecting information about accessed features, cache behaviors, miss reasons and optimizations objects. These tools have great performance analysis, but they do not address profiling and deep analysis of RAM memory in the same way as mVizion does.

A call-graph tool [3, 4] collects information about static as well as dynamic function calls, and presents it in a single hierarchical view. This tool is based on the underline Valgrind application, more specifically its extension Callgrind used for function call logging taking place during the application run-time. It is able to collect all available information about each function including its name, namespace and its class. This tool closely matches mVizion Stack Trace window since it collects and builds a function call tree. While the call-graph tool represents a wider set of function calls executed by the application and their hierarchical view, the tool doesn't maintain information about memory management nor report any type of statistics describing which data structures within the applications are bounded to which functions.

2.3 Java application visualization

Heapviz [2] and dynamic performance analysis (DYPER) [13, 1] are frameworks designed for visualizing and analyzing Java program performance during run-time. Heapviz not only provides a clean visual representation of how the heap is being used by a variety of large software applications, but it also does summarization techniques to combine similar objects into a single node hierarchy based on the data structure topology in place. A single node in the visualization can represent a large data structure belonging to a linked list, a tree map, etc. The visualization provides a graph hierarchy of nodes which is used to identify the sharing of data types and objects across application components. Even though the tool has a search option, it does not provide any memory layout visualization nor more detailed description of each object. Heapvizs goal is to understand and debug tasks rather than analyzing memory utilization, which is where mVizion presents a unique set of features.

The dynamic performance analysis (DYPER) [13, 1] tool was developed to analyze long-running systems in a production environment. While having minimal overhead on the running system it is capable of providing detailed information about CPU activity, heap utilization, I/O behavior, sockets, threads, memory allocations, event handling, as well as phase analysis. DYMEM [14, 1] is an extension to the DYPER framework providing a visualization of object ownership in the heap as well as the memory utilization of an application. Both DYPER and DYMEM are not able to evaluate the entire program run-time but rather analyze a slice of run-time. Furthermore, scanning the heap can cause the program to pause during the execution phase depending on the size of the heap.

2.4 C/C++ application visualization

GCspy [11] is another architectural framework designed for collection and replay of memory management behavior of large-scale systems running Java programs. The framework is able to attach and detach from running applications by using a memory manager. An extension to GCspy [7] has been developed to track the dynamic memory allocators and heap usage in C/C++ programming applications. The extension tool modifies dlmallocs [10] source in order to collect information about each malloc and free events of a memory manager. Both GCspy and its extension are designed to keep track of heap states and its rendering in response to both garbage collection and memory manager events. This aspect imposes limitations on the amount of information being collected affecting the overall functionality and granularity of the tools.

A different approach to memory performance analysis was proposed by AllocRay [16], a visualization tool capable of animating the memory allocations and usage over the complete period of application execution. AllocRay is able to collect heap allocations/de-allocations, re-allocation, free events with matching timestamps, detailed information about object types, addresses, sizes, processes, as well as function call stack traces for all events. The collected information is then replayed as a visualization with multiple interactive options including playback, zooming, color coding and filtering. Even though AllocRay shares many similarities with mVizion, there is a fundamental difference between both of them. AllocRay is specifically designed to operate in a Windows environment and works closely with Microsofts Event Tracing for Windows (ETW) library. On the other hand mVizion works closely with Cafegrind [5] as well as Processing, allowing it to operate in multiple environments.

Chapter 3

Design and Implementation

3.1 Visualization interface and functionality

mVizion (memory Visualization) tool has been implemented on top of Processing framework [9]. Processing is a Java based open source programming language which includes graphics, network, PDF, JavaScript, and other libraries; and provides an environment for sketching programmatical visualizations. The Processing framework is supported in Windows, GNU/Linux, and Mac OS X platforms and is portable to web applications.

Processing works both in 2 and 3 dimensions but is limited to the most basic geometric shapes. mVizion works in the 2-D vector space and utilizes only 3 simple shape structures, which include rectangles, lines and arcs. Overlaying multiple virtual matrices on top of the visual canvas with simple trigonometrical calculations makes it easier to divide the window 6 ways, each one of them processing and presenting, using smooth animations and various color shades, a separate set of data throughout all the visualization. Processing also features support of buttons, scroll-bars and mouse events (clicked, pressed, moved, etc).

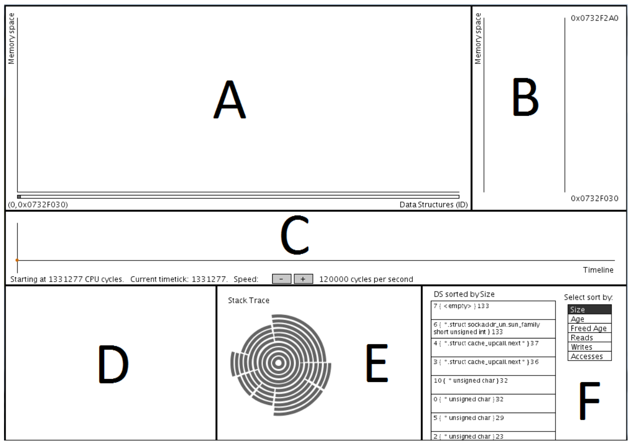
mVizion takes advantage of object-oriented programming as it is based on Java programming language. The tool consists of 5 distinct classes, each one of them containing all necessary information collected by Cafegrind describing each of the data structures belonging to the target application. This information is populated into several fields of mVizion internal data structures including object data fields, time event sequence, function trace, and sorted by size, age, freed age, reads, writes and access operations. The aggregated data is loaded in a setup phase of mVizion run-time. The user selects three data files (summary file, freed file and heap file), which mVizion takes as an input to populate the necessary information into its internal data structures (not to be confused with the target application data structures).

The summary and freed log files contain detailed information about each application data structure consisting of several fields such as unique IDs, data types, memory addresses, requested sizes, extra bytes, timestamp of when they were created, time when they were last accessed, number of reads and writes, overall age, freed age (age after data structure has been de-allocated until its memory space is overwritten), and allocation function stack trace. This data is entirely loaded into memory using different Java components such as arrays and ArrayLists.

In a similar fashion the heap file contains information split into several fields including each load/store operation as well as the address at which the operation is being performed, the beginning address of the data structure, its unique ID, the size and timetick of the access operation.

mVizion uses its internal data structures and a temporary file to load large log files. The tool stores in memory only target application object-related information while saving into a file a timeline sequence of access operations in order to reduce the memory usage. This file is read line by line during the visualization display phase. All the complex computations are done in mVizion setup phase thus reducing the overhead during its execution.

The visualization is done during the draw step of mVizion display phase. Processing draw methods are executed repeatedly after the previous method invocation finishes with a specified delay between each invocation. This delay can be easily adjusted and is specified in Processing language through a rate of frames per second where each frame is a single execution of a draw method; the default mVizion rate is 60 frames per second. This frame rate can be significantly slowed down during mVizion display execution as it primarily depends on the time the previous method invocation finishes. In order to adjust the speed of the visualization mVizion uses an additional variable that specifies the time step in the timeline execution. The timeline represents the CPU cycles of the target application run-time and contains both the access operations as well as their respective timestamps in sequential order. The timetick step value specifies the number of CPU cycles mVizion executes in each frame. This measure is able to speed up the application run-time in case there are large gaps between many adjacent access operations, but it can also have a reverse effect and create a visible impression of the application slowing down as the execution enters a loop.



```
Starting at 1331277 CPU cycles.   Current timetick: 1331277.   Speed:   120000 cycles per second
```

```
7 { <empty> } 133
6 { *.struct sockaddr_un.sun_family short unsigned int } 133
4 { *.struct cache_upcall.next * } 37
3 { *.struct cache_upcall.next * } 36
10 { * unsigned char } 32
0 { * unsigned char } 32
5 { * unsigned char } 29
2 { * unsigned char } 23
```

Figure 3.1: mVizion canvas

As presented in Figure 3.1, mVizion visualization canvas is comprised of six sections: (1) the data structures window; (2) the heap window; (3) the timeline access window; (4) the object data window; (5) stack trace window; and (6) Dynamically Sorted Object List window.

3.2 Data structures window

Data structures window (part A in Figure 3.1) presents an explicit visualization of data structures versus their location in memory space with respect to other data objects and representative of each object's size and active/freed status during the run-time. The data being loaded from log files serves as a basis for further analysis and extraction of latent variables. mVizion keeps track of such variables and their dynamic values during execution. On the other hand it infers information about the multiple data structure re-allocations during it's lifetime by aggregating the necessary data into a minimal number of memory blocks and presents the resulting information onto its respective location within the graph.

The goal of aggregating the data and its graph representation is to (1) evaluate the efficiency of memory space usage as well as (2) further analyze the security implications of memory re-use and memory overwriting. Side-by-side view of each data structure shows patterns of how new data structure objects get allocated and de-allocated in time. The visual representation of block sizes and their active lifetimes provides rich information regarding the frequency ratio of memory usage of addresses versus the size of the data objects being stored in these addresses. The size of each data structure is computed considering its actual size and scaling it using a fixed normalization factor. In the case where the size of an object height is rounded to zero, it automatically defaults to 1 pixel value. The minimal width of each data structure is set to be 5 pixels.

On the other hand this window contains a horizontal scroll bar that enables the possibility of scrolling through the entire data object space allowing the user to further analyze data structures that do not fit into the snapshot view of the window. When the number of data structures present is less than the allowed horizontal pixel space, the scroll bar is non-functional.

Every single data structure is visualized in this window independent of its content or the number of accesses performed on the data object. One special case encountered during the implementation of mVizion occurred when the data structures were created but never written to during their lifetime or have no type. Nonetheless, the minimum information about this type of data structures still exists in the log files in order to be able to display the object information in this window.

This window represents each data structure as a rectangle of a minimum size of 1x5 pixels or greater. Each rectangle supports different actions including mouse rollover and mouse press events including cases when the scroll bar is used. The color scheme for each data structure consists of 3 different shades of gray. Medium gray denotes the active data object; light gray represents data structures that have been previously de-allocated; and dark gray is used consistently throughout the 6 windows of mVizion to highlight the selected (rolled-over or clicked) data object and its respective information. Additionally the red color is used at the end of mVizion execution to denote which data structures have not been de-allocated (memory leaks). Furthermore the blue color is used to denote which data structures have been de-allocated but their memory address space hasnt been overwritten.

3.3 Heap window

The heap window (part B in Figure 3.1) is a complimentary visualization to the data structures window. It is a combined view of all data objects visualized on top of one another in the same memory space layout as it was in the data structures window. The same color scheme is used. This visualization window also supports mouse rollover and mouse pressed events in the order of the latest allocation in a particular memory address pixel. This window includes labels denoting the initial memory address and the last memory address.

3.4 Timeline access window

mVizion constructs an internal timeline data structure for representing a sequence of events and operations of the application data structures (part C in Figure 3.1), which include allocation, de-allocation, storing, and loading operations. This information found in the summary and heap log files is stored in a file in a sequential order of events. During execution mVizion reads line by line this temporary file and executes each operation in accordance with the type of event.

The Timeline access window is an arc diagram visually representing the lifetime of each data object after its de-allocation. mVizion keeps track of all allocation and de-allocation operations and uses this information to represent the lifetime age of freed data objects by using arcs. An arc diagram gives a visual view of the sequence of events and allocation/de-allocation patterns throughout the application execution.

The algorithm for populating the timeline of events is based on a pre-sorted array list and the insertion of each new element into the list in sequential order. From the summary and the freed log files, the data about each allocation and de-allocation accompanied by their respective timestamps is inserted into Timeline data structure using a binary search algorithm. The running time for reading timestamps of allocations and de-allocations in a random order and inserting them into the array is logarithmic time. On the contrary, heap log events are inserted into the array using a separate method with the intention of reducing the running time even further. The access events are logged in a sequential order and thus can be easily inserted into Timeline following the same order. In order to achieve this mVizion monitors the current index in the array list and adds a new element in the next available slot as long as it satisfies the increasing ordering condition. This action speeds up mVizion setup phase execution by reducing the insert operation to approximately constant running time.

Timeline access window is only responsive to the mouse rollover and mouse clicks performed in data structures and heap windows. The arc corresponding to the selected data structure is highlighted in bold dark gray color on the diagram as long as that object has already been freed. Additionally a blue color is used at the end of mVizion run-time to denote which data structures have been de-allocated but their respective memory address spaces have never been overwritten.

This window also contains the labels with the starting timetick in CPU cycles, the current timetick, and the speed at which the visualization is running. The speed can be adjusted (increased/decreased) according to the user's needs using two buttons, each one labeled with plus/minus signs. The default speed is set to 12000 CPU cycles per second (cps) and is changed by 6000 cps each time the button is pressed with the lowest available value being 0 cps which pauses mVizion execution. Moreover at the end of mVizion run-time a new label with the total number of memory leaks is shown at the top of the window in red color.

3.5 Object data window

The object data window (part D in Figure 3.1) appears only in response to mouse events either on the data structure window or the heap window. On a mouse rollover event the view is only displayed for the time a mouse remains within the data structure boundaries, while in the case of a mouse-pressed event the object data window will remain showing.

Once a data structure has been selected its internal view, current status and other internal data are displayed in this window. The internal view represents the number of re-allocations of a particular data structure. This information is presented with the address space growing from the bottom up following from the same behavior as portrayed in the data structure and heap windows. The size of the data object representation is calculated from the size of the allocations scaled by a normalization factor. Depending on the size of the data structure each individual memory address is bounded to 5 pixels per representation for small-sized blocks, and in other cases each pixel can have an aggregation of multiple memory addresses for large allocations. Each data object block is evenly divided into such memory space representations which are then used to show the number of accesses performed on each memory space following a specified coloring convention, from light gray corresponding to no accesses to dark gray to black color representing the higher number of reads and writes.

The rest of information about a data structure is displayed in a text form and can be divided into static and dynamic data. The static data includes the data object ID, its type, an address array containing all of the allocations during its lifetime and their corresponding requested and extra sizes also in array forms, time created, lifetime age, and freed age. Freed age is defined in Cafegrind and it represents the data from a de-allocated object residing in a memory until it is overwritten by a new data structure. The case when a new data structure has been allocated in the same memory space as original object but never was overwritten is considered to be an exception and results in the original object's freed age defaulting to a value of -1. The lifetime age field denotes the number of CPU cycles the data structure has had until its de-allocation, or is set to -1 for objects that are never freed from the memory. The age field is an indicator of data persistence as well as memory leaks.

Dynamic object data fields include the time when it was last accessed, the number of reads and writes, and its current de-allocation status. This information is collected during mVizion execution. As the timeline sequence of events advances the data fields are automatically updated as needed. The number of reads found in the heap log file and collected into the timeline data structure is equal to the total number of reads provided in the summary log by the end of data object lifetime. The total number of writes are computed from the number of stores logged in the heap file plus the number of re-allocations for that particular data structure.

3.6 Stack trace window

The stack trace window (part E in Figure 3.1) is a ring diagram representation of the function stack tree data structure. The stack trace information collected from the summary log file represents the recursive stack of functions called to allocate a particular data object. This information is used to construct a tree data structure rooted at the application ”main” function or functions ”below main” and branch out leading to allocation function calls (”malloc”, ”calloc”, etc.) for all data objects.

The stack trace tree contains in each node the function name and the number of data structures making a call to that particular function. This number is used internally to compute the degree angle of an arc in a ring diagram reflecting the frequency of a function encounter. Since the ring diagram definition bounds the circumference of each level in a tree to add up to exactly 2*PI, this creates a visual limitation to the stack trace window. The tree branches with a small number of calls to these functions are limited to a very small degree angle making it practically impossible to represent them using a pixel on the canvas.

The ring diagram is always displayed in the stack trace window, but the full-featured view is only visible when any data object of the application has been selected. This view includes a data object's function stack branch highlighted in a dark gray color and a display of a function trace listing to the right of the diagram.

The stack trace tree is built incrementally, starting with the first available function stack and is appended with new function traces as needed. Each trace contained in the logs includes up to 8 function calls always starting with an immediate allocation function call that is represented as tree leafs in the internal data structure. The algorithm searches through the entire tree until it finds a match with any of the functions in a particular trace. After the match is found mVizion traverses up and down the trace to match the rest of the functions or create new nodes as needed. This representation is limited by the stack trace length when the listed 8 functions can't be matched with any of already existing functions in the tree. In this case, this trace will be omitted from further analysis and will not be considered a part of the tree.

3.7 Dynamic object list window

In the dynamic object list window (part F in Figure 3.1) mVizion presents a list view of the present data structures sorted by a selected filter. During the execution mVizion collects and maintains 6 internal lists of ordered data objects sorted by the data structure size, age, freed age, number of reads, writes, and total accesses. Each field from the list contains 3 values: the data structure ID followed by its type and the selected variable value. These lists are dynamically updated during the execution of timeline events where the top 8 data objects are displayed at all times.

This window enables the user to choose a particular filter to sort the list. This feature works independently of mouse rollover and mouse pressed events, allowing the user to keep previously selected data object in focus while changing different filter. mVizion internal lists can be divided into static and dynamic. Size, age and freed age object lists are populated statically during the setup phase, while the number of reads, writes and total accesses are collected and updated dynamically during the display of the timeline events.

Moreover this window is responsive to mouse events coming from data structures and heap windows. Whenever a data structure is selected it is highlighted in dark gray. It is important to note that it will only be visible in the case when this particular data object is present in the list. Since the data structures are being constantly updated, the user might see the data objects changing their places on the list based on selected filter during the execution.

Chapter 4

Evaluation

4.1 General

mVizion processes and visualizes the data log files provided by Cafegrind. mVizion and Cafegrind are not directly connected therefore the user must select the log files to be loaded into the visualization. The performance of mVizion is inversely proportional to the size of these files. Running most generic programs with Cafegrind results in a set of summary/freed/heap log files that vary in size. While the summary and freed log files contain the same number of lines as there are data structures allocated during the run-time, the heap file contains all the read/write operations with their corresponding information thus increasing both the number of lines as well as the file size. During the execution mVizion, in order to optimizing its memory consumption, uses a temporary file to store sequential data found in both heap and summary log files. On the other hand, the processed data from both the summary and freed log files resides in memory during the entire mVizion run-time and furthermore is appended with additional tracking information collected dynamically from the access events. The size of the internal memory used by the end of mVizion execution grows approximately to 3-4 times the size of the original summary file (Figures 4.10, 4.15, 4.11, and 4.16). While mVizion is able to process large log files, it's important to consider the sizes of the summary, freed and heap log files independently as well as the combined size of all three logs to estimate the growth and future size of mVizion internal data structures. There exists the possibility that mVizion will stop its execution if the memory utilization surpasses the available memory allocation.

mVizion has ran on four different machines:

- Machine A: Intel Core 2 Duo T9300 CPU at 2.50GHz 32-bit Windows 7 OS laptop with 4.0GB (3.0GB usable) RAM;
- Machine B: Intel Core i7 920 CPU at 2.67GHz Gentoo Linux 1.12.14 2.6.34 r12 kernel x86 64-bit desktop with 6.0GB RAM;
- Machine C: Intel Core 2 Duo E8500 CPU at 3.16GHz 64-bit Windows 7 OS desktop with 8.0GB RAM; and
- Machine D: Intel Core 2 T7400 CPU at 2.16GHz 32-bit Windows 7 OS laptop with 2.0GB RAM.

Before running mVizion, the Processing environment variables were set to reserve 1024MB of RAM for the most optimal operation, and all external processes were stopped.

Cafegrind was used along with Links, the Linux text web browser, in order to generate the log files for the evaluation section. Links application connected to www.facebook.com using two different networking protocols: HTTP and HTTPS.

mVizion was tested using both log sets on all 4 machines. Its run-time was divided into 2 logical phases: (1) the setup time and (2) the display time, where each phase was divided into a set of checkpoints at which performance statistics were collected. These statistics include the number of lines read from a heap file, number of lines written to a timeline file, number of CPU cycles performed, time taken, and size of the memory dump. A memory dump in this evaluation is an explicit write operation of all mVizion internal data, currently stored in RAM, into a file. Performing a memory dump and measuring the size of the output file provides a rough estimate of how much RAM mVizion is using at a particular instant.

4.2 Phase 1: Setup

Loading the log files takes a significant amount of time during mVizion execution, which is distributed across different files according to their sizes. Table 4.1 along with Figures 4.1, 4.2 and 4.3 show how over 98% of the time is spent processing the heap file data, this includes reading the data and writing it to a timeline file. Furthermore, Figures 4.4, 4.5 give a break down of time spent reading each 100,000 lines from a heap log file. The two graphs represent the different loading times across all 4 machines using both HTTP and HTTPS protocols. From Figures 4.4, 4.5 and Table 4.1 it is evident how the hardware resources and, more specifically, CPU power of the available computer can significantly speed up mVizion setup phase. RAM characteristics of the computers have no affect on mVizion performance in this case since the Processing maximum available memory (1024MB) is always below the hardware capacity. An interesting observation is that all 4 curves on both graphs slowly decrease as the amount of processing time spent in each period decreases between checkpoints. From mVizion implementation stand-point there is no clear explanation, but it could be potentially explained by the speed-up of data caching on the read and subsequent write operations.

mVizion builds the internal data structures as it reads the log files, beginning with summary file. It populates the timeline internal structure with allocation and de-allocation events found in a summary file. Only when mVizion execution reaches the point where it starts processing heap log, the tool begins writing the events in their sequential order to a temporary timeline file. It is for this reason the number of lines written to a timeline file varies over time from the number of lines read from a heap file. Figures 4.8 and 4.9 show how the number of writes per each epoch

machine	HTTP summary file	HTTP freed file	HTTP heap file	HTTPS summary file	HTTPS freed file	HTTPS heap file
A	0.403	0.5	120.297	0.694	0.785	189.192
B	0.5	0.303	47.846	0.713	0.544	80.064
C	0.345	0.365	98.678	0.545	0.631	154.324
D	0.489	0.577	135.273	0.8	0.955	214.733

Table 4.1: Load time in seconds per different machines

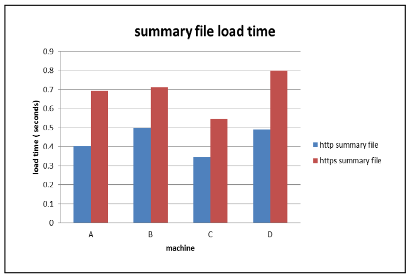


Figure 4.1: Summary File Load Time

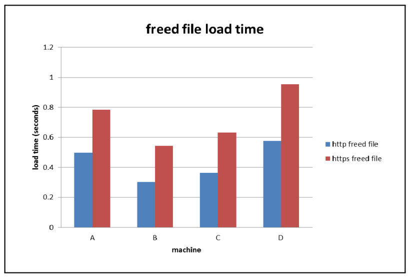


Figure 4.2: Freed File Load Time

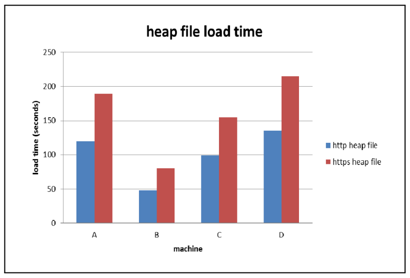


Figure 4.3: Heap File Load Time

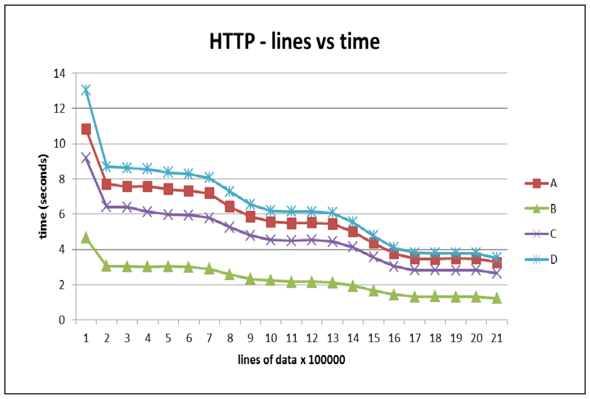


Figure 4.4: HTTP protocol Lines vs Time

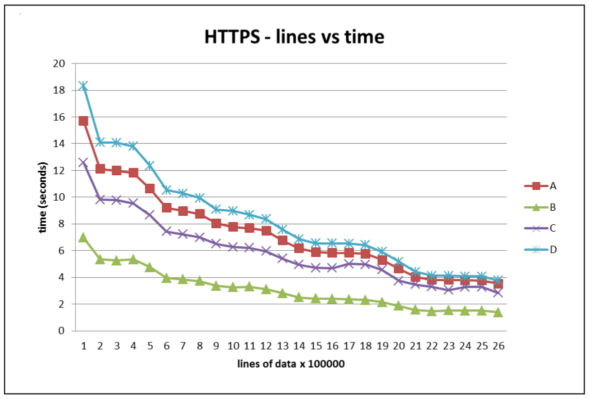


Figure 4.5: HTTPS protocol Lines vs Time

varies throughout the run-time. The noticeable variations at the beginning and the end of the graphs represent how there are many allocations and de-allocations, respectively, while for the rest of executions these numbers are evenly distributed. Figures 4.6 and 4.7 represent the progression of the number of lines written. They start at a significant number of allocations and de-allocations, but slowly grow over time until they reach their maximum peaks. These maximum peaks of written lines represent the total number of allocations and de-allocations for the entire application run-time, and is equal to twice the number of data structures created during the application execution minus the number of memory leaks.

Additional performance statistics include the memory dumps collected during the setup phase. They show the rate of memory utilization growth as more data is loaded into mVizion internal data structures. Figures 4.10 and 4.11 show this progression across HTTP and HTTPS protocols, and Figure 4.12 merges the previous two graphs into a single

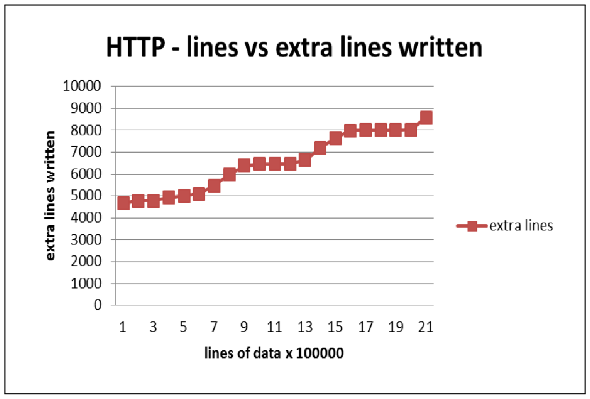


Figure 4.6: HTTP Lines vs Extra Lines Written

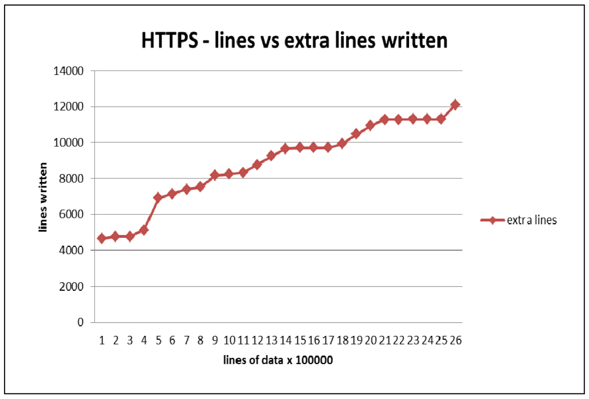


Figure 4.7: HTTPS Lines vs Extra Lines Written

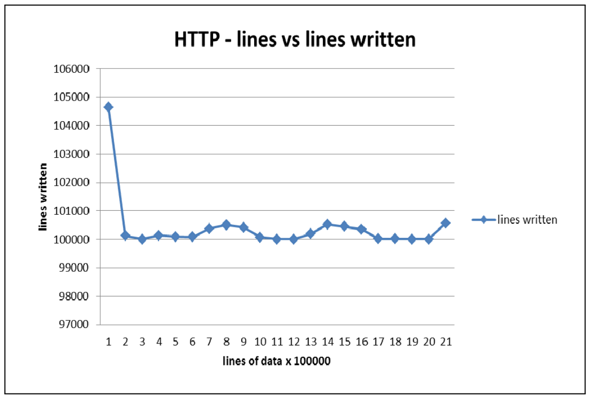


Figure 4.8: HTTP Lines vs Lines Written

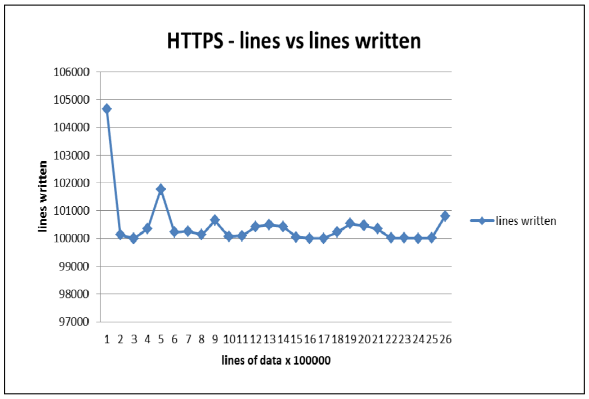


Figure 4.9: HTTPS Lines vs Lines Written

composition. From Figure 4.12 it is noticeable how HTTPS grows faster and includes more data points. Hence it can be concluded that HTTPS, when compared with HTTP protocol, contains more data structures and access points. This is congruent with the way HTTPS secures its data. On the other hand, both plots share a significant number of identical data points. This shows how similar both protocols are when being executed in a browser-type application.

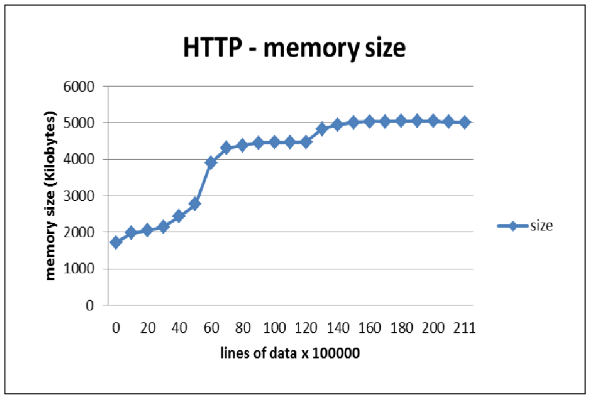


Figure 4.10: HTTP Lines vs Memory Size

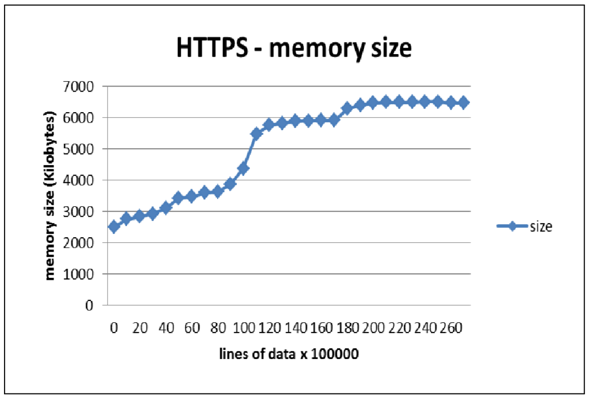


Figure 4.11: HTTPS Lines vs Memory Size

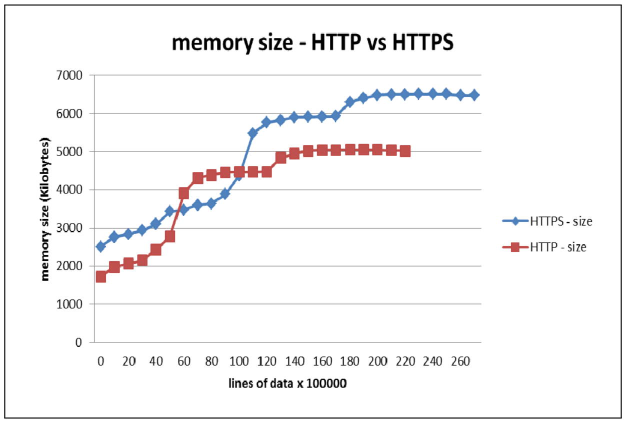


Figure 4.12: Memory Size: HTTP vs HTTPS

4.3 Phase 2: Display

Most of mVizion run-time is spent rendering the canvas along with the animation. The length of the timeline as well as the number of access operations previously loaded from the summary and heap log files determines the length of the visualization display and is deterministic for the same set of logs. Figures 4.13 and 4.14 show the time taken by each machine to process the same logs over the equal set of time intervals. The determinism of these events causes mVizion performance to be proportional to the CPU power of the machine. Figures 4.13 and 4.14 also show a slow down in mVizion execution over time that is consistent across all 4 machines, and can be explained by a continuously growing number of timeline events as well as the size of internal memory utilization.

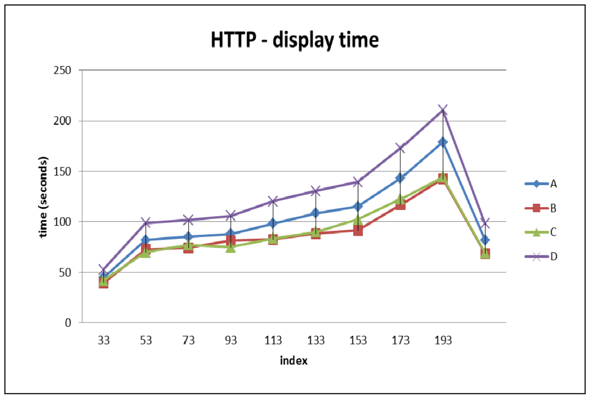


Figure 4.13: HTTP Display Time

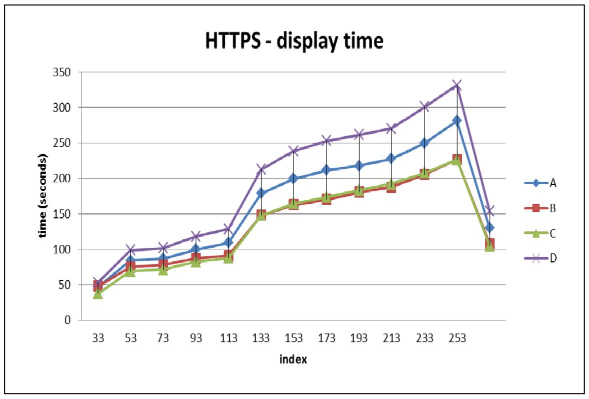


Figure 4.14: HTTPS Display Time

Another performance metric obtained during mVizion display is the size of memory dumps. The size of the internal RAM utilization measured with periodic memory dumps shows a significant increase in memory usage. This growth continues throughout mVizion run-time, both in the setup phase and the display stage, as portrayed in Figures 4.15 and 4.16. As mVizion reads the access events from the timeline file, it updates and appends the new data to mVizion internal data structures thus increasing the memory usage. As seen in Figures 4.10, 4.11, 4.15, and 4.16 the overall growth throughout mVizion run-time is approximately 3-4 times the original summary data set.

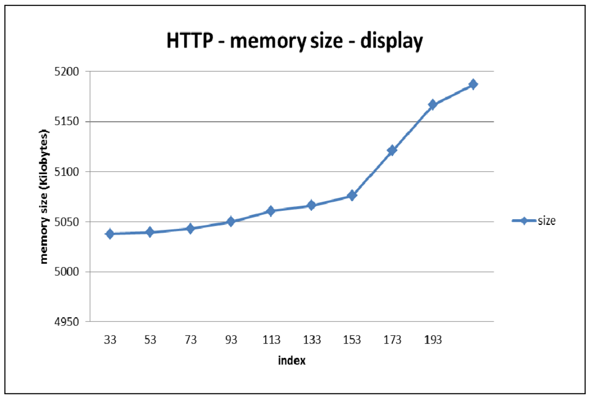


Figure 4.15: HTTP Memory Size Display

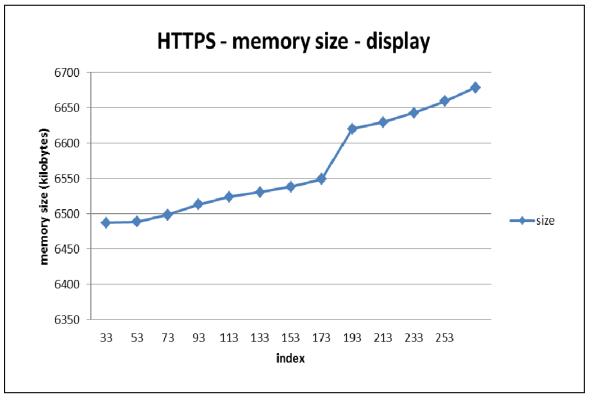


Figure 4.16: HTTPS Memory Size Display

Chapter 5

Future Work

mVizion was developed to run as a stand-alone application, but since it is implemented on top of the Processing framework it is feasible to convert the existing code into a web application. Furthermore, it is possible to develop multimedia components to create videos or snapshots during mVizion execution.

Additional functionality and feature support to be adopted in the future include the development of mVizion API. Currently mVizion is available only as a source code to be loaded into Processing framework. The future goal is to create an executable that automatically installs, runs Processing and plays the visualization without requiring any additional configuration setting from a user. The creation of mVizion API will allow programmatical access and scripting support to mVizion parsed and inferred internal data.

Further steps include improvement of mVizion running time and memory utilization. Since the maximum available memory to be used by mVizion is set to 1024MB, this limits the capabilities of its speed as well as processing power. In order to improve the robustness of the tool it will be necessary to analyze in more detail what data is being duplicated across mVizion internal data structures and consider implementing either novel compression algorithms, or write least frequently used data to temporary files that can be accessed during the run-time. Furthermore, the goal for the next revision of mVizion is to make use of all the data and logs Cafegrind offers including type accesses, extend the functionalities of the windows, and display more granular information. Other considerations include integrating canvas plots into a single mVizion window with multiple tabs and/or buttons in order to select the view, granularity and zoom factor.

There are several low level implementation details that have been omitted from the current mVizion design. Currently mVizion only handles multiple re-allocations for two or none overlapping data blocks. The case of 3 or more overlapping blocks is currently not supported. The general algorithm for multiple re-allocation support would include: (1) identifying the number of overlapping blocks; (2) sorting them in the order of memory addresses; (3) identifying where the partial or full block overlaps occur; and (4) aggregating the blocks in a particular order while maintaining the ordering, the addresses, the block sizes, and the current read/write statistics. Additionally, mVizion could maintain the information about each re-allocation independent of aggregated memory blocks. Other future work includes modifying the current color scheme and support different color shading in the Graph and Heap windows based on the number of accesses of each particular data object.

The type access log file is the last output provided by Cafegrind and is not supported in the current implementation of mVizion. This log file contains detailed stack traces of function pointers performing access operations on different data structures that can complement the current mVizion internal data structures.

Chapter 6

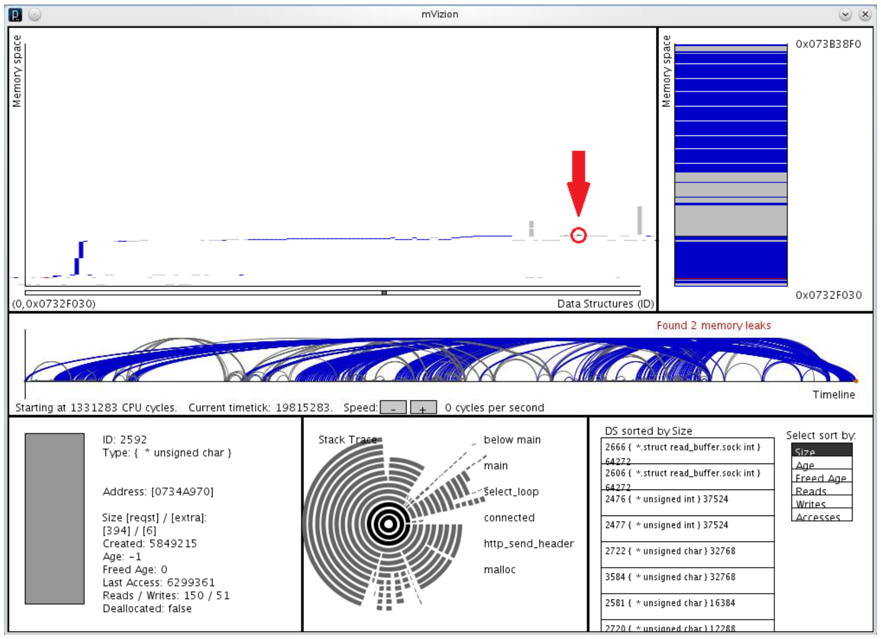
Conclusion

mVizion is a memory analysis and visualization tool for Cafegrind, a forensic profiling tool. It interprets the logs generated by Cafegrind and reconstructs the trace of events in the order of their occurrence. By aggregating and analyzing the extracted data mVizion can infer information that would not be possible to extract otherwise with other manual methods.

mVizion presents detailed information about each individual data structure and provides a number of various ways through 6 visualization windows to analyze and understand memory utilization behaviors. mVizion provides an appealing, simple and clear interface representing all allocation, de-allocation, read and write events in their sequential order and detailed information about each individual data object. Using different visual components it is relatively easy to detect unusual memory behaviour, observe possible memory bugs, churn, memory fragmentation, and locate memory leaks.

Appendix A

Visualization snapshots



Found 2 memory leaks

Starting at 1331283 CPU cycles. Current timetick: 19815283. Speed: - + 0 cycles per second

ID: 2592
Type: { * unsigned char }

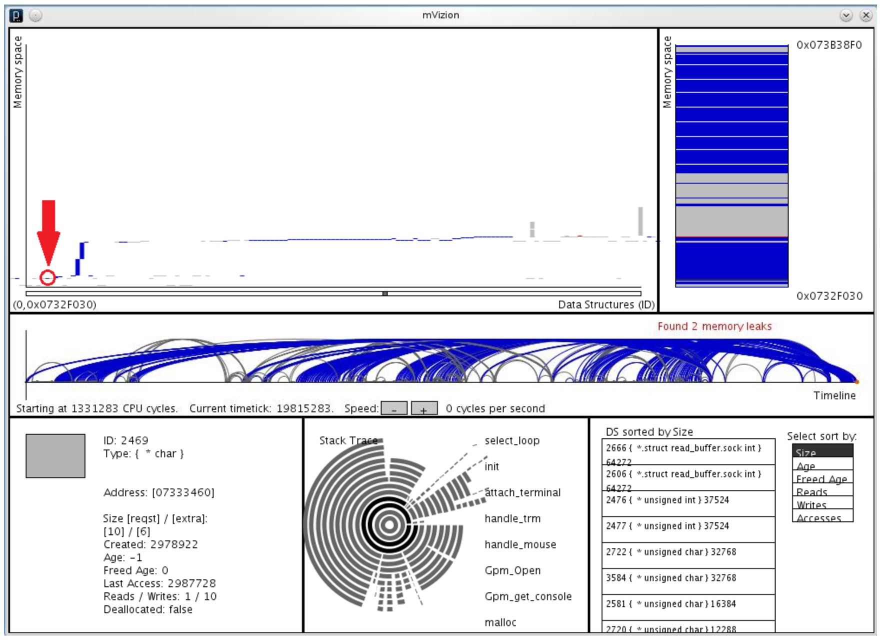
Address: [0734A970]

Size [reqst] / [extra]:
[394] / [6]
Created: 5849215
Age: -1
Freed Age: 0
Last Access: 6299361
Reads / Writes: 150 / 51
Deallocated: false

DS sorted by Size

```
2666 { *.struct read_buffer.sock int }
64272
2606 { *.struct read_buffer.sock int }
64272
2476 { * unsigned int } 37524
2477 { * unsigned int } 37524
2722 { * unsigned char } 32768
3584 { * unsigned char } 32768
2581 { * unsigned char } 16384
2720 { * unsigned char } 12288
```

Figure A.1: Memory Leak A found running Links application using HTTP protocol: data structure ID: 2592, type: { * unsigned char }, address: 0x734A970, size: 394 Bytes, reads/writes: 150/51, created: 5849215, last accessed: 6299361



Found 2 memory leaks

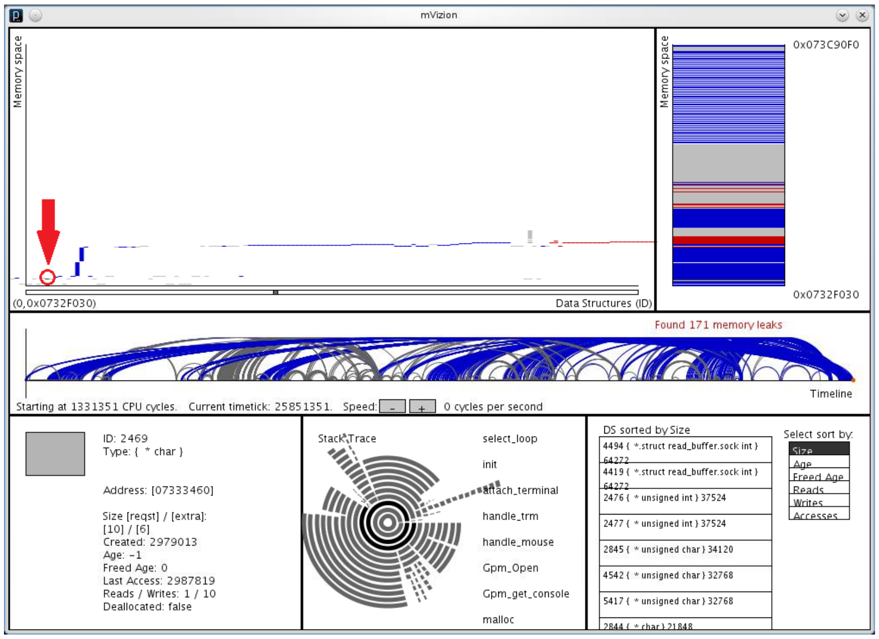
Starting at 1331283 CPU cycles. Current timetick: 19815283. Speed: 0 cycles per second

```
ID: 2469
Type: {  * char }

Address: [07333460]

Size [reqst] / [extra]:
[10] / [6]
Created: 2978922
Age: -1
Freed Age: 0
Last Access: 2987728
Reads / Writes: 1 / 10
Deallocated: false
```

DS sorted by Size

2666 { *.struct read_buffer.sock int } 64272
2606 { *.struct read_buffer.sock int } 64272
2476 { * unsigned int } 37524
2477 { * unsigned int } 37524
2722 { * unsigned char } 32768
3584 { * unsigned char } 32768
2581 { * unsigned char } 16384
2720 { * unsigned char } 12288

Figure A.2: Memory Leak B found running Links application using HTTP protocol: data structure ID: 2469, type: { * char }, address: 0x7333460, size: 10 Bytes, reads/writes: 1/10, created: 2978922, last accessed: 2987728



Found 171 memory leaks

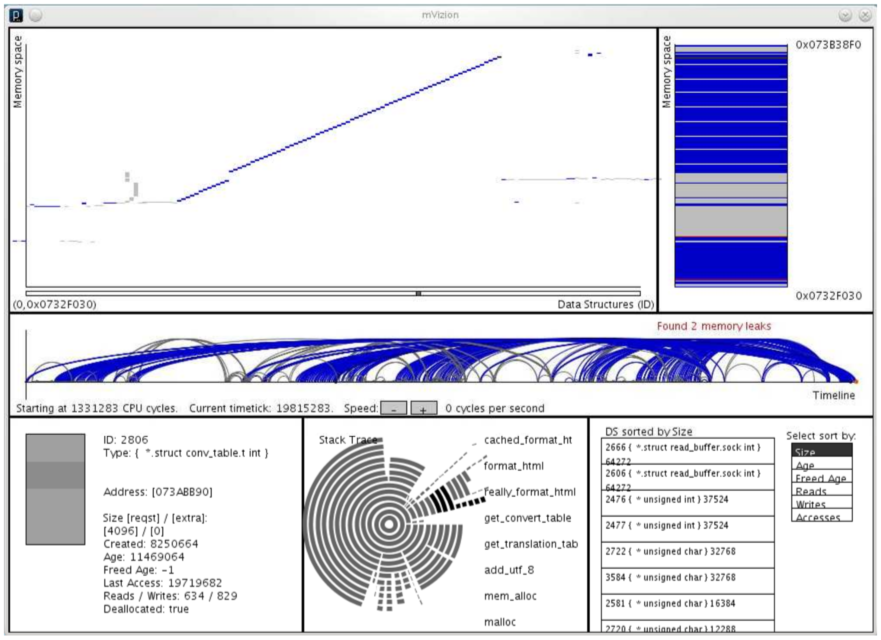
Starting at 1331351 CPU cycles. Current timetick: 25851351. Speed: 0 cycles per second

```
ID: 2469
Type: {  * char }

Address: [07333460]

Size [reqst] / [extra]:
[10] / [6]
Created: 2979013
Age: -1
Freed Age: 0
Last Access: 2987819
Reads / Writes: 1 / 10
Deallocated: false
```

DS sorted by Size

4494 { *.struct read_buffer.sock int } 64272
4419 { *.struct read_buffer.sock int } 64272
2476 { * unsigned int } 37524
2477 { * unsigned int } 37524
2845 { * unsigned char } 34120
4542 { * unsigned char } 32768
5417 { * unsigned char } 32768
2844 { * char } 21848

Figure A.3: Memory Leak B found running Links application using HTTPS protocol: data structure ID: 2469, type: { * char }, address: 0x7333460, size: 10 Bytes, reads/writes: 1/10, created: 2979013, last accessed: 2987819

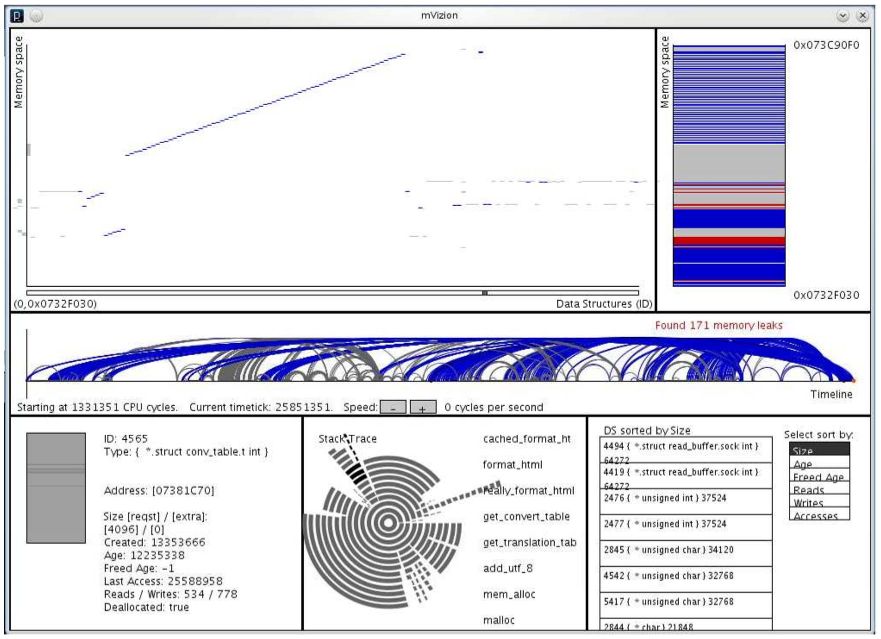


Starting at 1331283 CPU cycles. Current timetick: 19815283. Speed: - + 0 cycles per second

ID: 2806
Type: { *.struct conv_table.t int }
Address: [073ABB90]
Size [reqst] / [extra]:
[4096] / [0]
Created: 8250664
Age: 11469064
Freed Age: -1
Last Access: 19719682
Reads / Writes: 634 / 829
Deallocated: true

DS sorted by Size
2666 { *.struct read_buffer.sock int } 64272
2606 { *.struct read_buffer.sock int } 64272
2476 { * unsigned int } 37524
2477 { * unsigned int } 37524
2722 { * unsigned char } 32768
3584 { * unsigned char } 32768
2581 { * unsigned char } 16384
2720 { * unsigned char } 12288

Figure A.4: Memory Allocation Ladder found running Links application using HTTP protocol: data structure ID: 2806, type: { * .struct conv_table.t int }, address: 0x73ABB90, size: 4096 Bytes, reads/writes: 634/829, created: 8250664, last accessed: 19719682, age: 11469064



Starting at 1331351 CPU cycles. Current timetick: 25851351. Speed: - + 0 cycles per second

ID: 4565
Type: { *.struct conv_table.t int }
Address: [07381C70]
Size [reqst] / [extra]:
[4096] / [0]
Created: 13353666
Age: 12235338
Freed Age: -1
Last Access: 25588958
Reads / Writes: 534 / 778
Deallocated: true

DS sorted by Size
4494 { *.struct read_buffer.sock int } 64272
4419 { *.struct read_buffer.sock int } 64272
2476 { * unsigned int } 37524
2477 { * unsigned int } 37524
2845 { * unsigned char } 34120
4542 { * unsigned char } 32768
5417 { * unsigned char } 32768
2844 { * char } 21848

Figure A.5: Memory Allocation Ladder found running Links application using HTTPS protocol: data structure ID: 4565, type: { * .struct conv_table.t int }, address: 0x7381C70, size: 4096 Bytes, reads/writes: 534/778, created: 13353666, last accessed: 25588958, age: 12235338

References

[1] Dyvise. `http://www.cs.brown.edu/~spr/research/vizdyvise.html`.

[2] E. E. Aftandilian, S. Kelley, C. Gramazio, N. Ricci, S. L. Su, and S. Z. Guyer. Heapviz: interactive heap visualization for program understanding and debugging. In *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, pages 53–62, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0028-5. doi: http://doi.acm.org/10.1145/1879211.1879222. URL `http://doi.acm.org/10.1145/1879211.1879222`.

[3] J. Bohnet and J. Döllner. Visual exploration of function call graphs for feature location in complex software systems. In *Proceedings of the 2006 ACM symposium on Software visualization*, SoftVis '06, pages 95–104, New York, NY, USA, 2006. ACM. ISBN 1-59593-464-2. doi: http://doi.acm.org/10.1145/1148493.1148508. URL `http://doi.acm.org/10.1145/1148493.1148508`.

[4] J. Bohnet and J. Döllner. Analyzing feature implementation by visual exploration of architecturally-embedded call-graphs. In *Proceedings of the 2006 international workshop on Dynamic systems analysis*, WODA '06, pages 41–48, New York, NY, USA, 2006. ACM. ISBN 1-59593-400-6. doi: http://doi.acm.org/10.1145/1138912.1138922. URL `http://doi.acm.org/10.1145/1138912.1138922`.

[5] E. Chan, S. Venkataraman, N. Tkach, K. Larson, A. Gutierrez, and R. H. Campbell. Characterizing data structures for volatile forensics. In *Proceedings of the 2011 Sixth IEEE International Workshop on Systematic Approaches to Digital Forensic Engineering*, SADFE '11, Oakland, CA, USA, 2011.

[6] B. Chapma. Dragon analysis tool. `http://www2.cs.uh.edu/~dragon/Documents/User_Guide.pdf`.

[7] A. M. Cheadle, A. J. Field, J. W. Ayres, N. Dunn, R. A. Hayden, and J. Nystrom-Persson. Visualising dynamic memory allocators. In *Proceedings of the 5th international symposium on Memory management*, ISMM '06, pages 115–125, New York, NY, USA, 2006. ACM. ISBN 1-59593-221-6. doi: http://doi.acm.org/10.1145/1133956.1133972. URL `http://doi.acm.org/10.1145/1133956.1133972`.

[8] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 22–22, Berkeley, CA, USA, 2004. USENIX Association. URL `http://portal.acm.org/citation.cfm?id=1251375.1251397`.

[9] B. Fry and C. Reas. Processing. `http://processing.org/`.

[10] D. Lea. A memory allocator. `http://g.oswego.edu/dl/html/malloc.html`.

[11] T. Printezis and R. Jones. Gcspy: an adaptable heap visualisation framework. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 343–358, New York, NY, USA, 2002. ACM. ISBN 1-58113-471-1. doi: http://doi.acm.org/10.1145/582419.582451. URL `http://doi.acm.org/10.1145/582419.582451`.

[12] S. P. Reiss. Visualizing java in action. In *Software Visualization*, 2003. doi: 10.1145/774833.774842.

[13] S. P. Reiss. Controlled dynamic performance analysis. In *Proceedings of the 7th international workshop on Software and performance*, WOSP '08, pages 43–54, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-873-2. doi: http://doi.acm.org/10.1145/1383559.1383566. URL `http://doi.acm.org/10.1145/1383559.1383566`.

[14] S. P. Reiss. Visualizing the java heap. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 251–254, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: http://doi.acm.org/10.1145/1810295.1810344. URL `http://doi.acm.org/10.1145/1810295.1810344`.

[15] S. P. Reiss and M. Renieris. Jove: java as it happens. In *Proceedings of the 2005 ACM symposium on Software visualization*, SoftVis '05, pages 115–124, New York, NY, USA, 2005. ACM. ISBN 1-59593-073-6. doi: http://doi.acm.org/10.1145/1056018.1056034. URL `http://doi.acm.org/10.1145/1056018.1056034`.

[16] G. G. Robertson, T. Chilimbi, and B. Lee. Allocray: memory allocation visualization for unmanaged languages. In *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, pages 43–52, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0028-5. doi: http://doi.acm.org/10.1145/1879211.1879221. URL `http://doi.acm.org/10.1145/1879211.1879221`.

[17] J. Tao and A. Shahbahrami. Data locality optimization based on comprehensive knowledge of the cache miss reason: A case study with dwt. In *Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications*, HPCC '08, pages 304–311, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3352-0. doi: http://dx.doi.org/10.1109/HPCC.2008.7. URL `http://dx.doi.org/10.1109/HPCC.2008.7`.