Data Structure Analysis and Memory Bloat in Unmanaged Languages

Adrian Vinojčić
Johannes Kepler University Linz
Linz, Austria
adrian.vinojcic@gmail.com

Alexander Voglsperger
Johannes Kepler University Linz
Linz, Austria
alex@wtf-my-code.works

ABSTRACT

Memory bloat and memory leak are terms that apply to unmanaged languages such as C and C++. To find such problems, information about the used data structure is key. However, such an analysis is difficult because little information about the data structure exists after compilation. In addition, memory bloat and memory leaks can be difficult to find in larger projects and complex structures because they often emerge over time. This seminar paper presents some approaches for data structure analysis and ideas to detect and mitigate memory bloat and leaks.

KEYWORDS

Memory Bloat, Memory Leak, Unmanaged Languages, Data Structure Analysis

1 INTRODUCTION

Software development can be performed in various languages, with their own advantages and disadvantages. One aspect that applies globally to every language is Myhrvold's Law, which states, that *software is like a gas* and grows bigger and bigger the more space it gets [5] [10]. According to Lee, software complexity and size scale even faster than clock speed [5]. Developers have the tendency to push hardware boundaries and add functionality to their programs, as Xu et al. explain in their paper [10].

In order to deal with the growth of software complexity, one key feature of numerous managed languages is a garbage collector, which works in the background and continuously collects all unreferenced data from the heap at runtime.

In this paper, we focus on unmanaged languages, i. e., languages which do not provide garbage collector or other memory safety features, such as Rust's borrow-checker. Writing software in an unmanaged language requires the developer to responsibly allocate and deallocate heap space. Manual heap space allocation and deallocation has the advantage of possible optimizations and allowing the developers to write real-time applications, since their program will certainly not come to a halt due to a garbage collection. The great flaw with manual memory management is human error, that leads to memory mismanagement in the form of memory bloat and memory leaks. Memory bloat and memory leaks will be explained in more detail in Section 2. Memory mismanagement is often difficult to detect, because the symptoms are observable very late in testing. Furthermore, diagnosis may be difficult, since the final allocation that brings the program to crash in the end, needs not to be related to the error, but it is just the final straw [1].

For clarity, we find it useful to distinguish between different forms of memory mismanagement using the definitions used by Hill et al. [1]. We refer to a memory leak, when all references to a memory block are lost and the address space can not be deallocated anymore. To distinguish from a subtype of the common definition of a memory leak, we introduce the term memory cyst for a block of memory, which is referenced, but no longer used in the rest of the program. These blocks sit stale in memory. Other important candidates for memory mismanagement are *aggregates*, i. e. data structures, such as lists, trees and maps. When these aggregates grow without bound, we call them memory tumors. Memory tumors usually grow until the program crashes at some point.

In this paper, we compare solutions to different causes of memory bloat. Furthermore, we discuss some approaches to detect and fix memory bloat, as well as an idea inspired by managed languages.

2 MEMORY BLOAT AND LEAKS

Memory bloat describes excessive use of memory, which typically manifests itself in the form of memory leaks, as pointed out by Xu et al. [10] and Lee [5]. Bloat can also arise from poor design and implementation, which could be handled more efficiently. The resulting memory footprint and slowdowns are notoriously difficult to debug and fix because bloat typically accumulates in long-running applications, as noted by Novark et al. [7] and Rahul et al. [9].

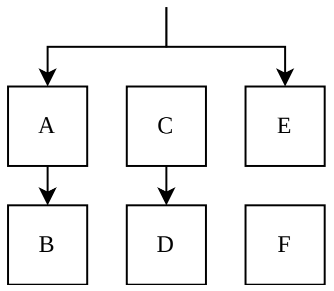

Figure 1: Visualization of memory with unreferenced blocks.

Rahul et al. divide memory leaks into two categories, where either the dynamically allocated memory becomes unreachable or the allocated memory is not freed despite the memory not being needed any more [9]. Figure 1 shows unreachable blocks *C* and *F* floating around in memory. These unreferenced memory locations may also reference other memory blocks that are not freed, as is the case with *C* referencing *D*. Figure 2 shows memory blocks that are still accessible but are stale, which Hill et al. call *memory cyst*. The memory blocks *C* and *E* are stale and could be released, which could also free blocks *D* and *F*, since both blocks are directly or indirectly referenced by block *C*.

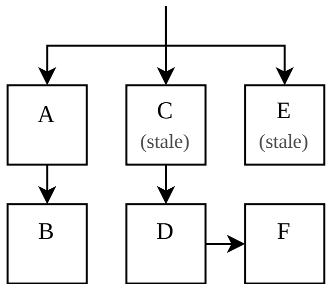

Figure 2: Visualization of memory with stale blocks.

Although developers try their best to manage memory properly, such problems can still occur as the code base grows, data structures are becoming increasingly complex and more dynamically allocated memory is used. Therefore, the following sections discuss different approaches to find and fix such memory problems.

Section 3 will discuss various static and dynamic detection systems that can be used to find, validate and fix such memory bloat problems. Section 4 focuses on the detection of data structures that are classified as memory tumors. In addition to detection and validation, there are also approaches to tune or replace the existing memory allocator to reduce or mitigate memory bloat, as discussed in Section 5.

3 DETECTION AND VALIDATION OF MEMORY BLOAT

Detecting memory bloat in applications is not a new concept, in fact there are many approaches to finding problems related to bloat or leaks. Contrary to managed languages, runtime information for such unmanaged languages is often limited. Consequently, this limitation is making it difficult to know which binary data represents what. These approaches typically fall into one of two categories. They either scan the code statically for specific ineffective patterns, or they monitor the memory and its data at runtime.

3.1 Dynamic Validation of Static Leak Warnings

Static analysis is widely used to find leaks in unmanaged languages such as C/C++. Li et al. mention that this type of analysis is capable of detecting all potential leaks in a program, with the drawback of reporting a great number of false positives. Because of this, finding the actual leaks is a daunting task and consumes a non-negligible amount of time on warnings that can be ignored. This time could be spent fixing the actual leaks or performing other important tasks [6].

For this reason, Li et al. developed a technique that dynamically generates tests from static analysis warnings and executes said tests to check their validity. The technique then classifies every warning into exactly one of the following four categories:

- *MUST-LEAK* – guaranteed leak
- *LIKELY-NOT-LEAK* – high likelihood to be a false warning
- *BLOAT* – Likely no leak, but should be fixed
- *MAY-LEAK* – No classification possible

The results are not formally proven and are therefore not guaranteed to be correct, but they still reduce the amount of false positives with a certain amount of confidence. Therefore, the classification allows concentration on MAY-LEAK classifications, as it could not be determined if these warnings show actual leaks. Nevertheless, this is often a significantly smaller subset of the original warnings [6].

3.2 Runtime Data Sampling

Another approach is sampling data during the runtime of an application. This allows detection of bloat that only occurs during runtime and cannot be detected due to complex structures, and can also be used to monitor memory usage over time.

However, creating periodic memory dumps is quite expensive in terms of storage bandwidth and capacity, so tracing is typically used. *Valgrind* and *Purify* are two existing tracing-based tools that track data movement of applications. However, these tools have a major flaw that makes them impractical in a production environment.

Novark et al. mention that these tools cannot locate reachable memory leaks and bloat. In addition, they say that these tools typically come with a huge overhead of up to 100×, which prevents their use in deployed applications [7].

Rahul et al. [9] and Novark et al. [7] present a type of Memory Leak Detector that provides a middle layer that sits on top of the default memory management of the language or the operating system. This middle layer then tracks certain data depending on the implementation.

The algorithm proposed by Rahul et al. internally creates a structure database and an object database which represents a graph-like structure. The algorithm is then able to analyze the generated graph to find unreferenced objects and report them as leaks. Figure 3 has been adapted from the original paper and shows the graph with the detected categorization [9].

In addition to the graph-based approach, there are systems such as *Hound*, which is a system proposed by Novark et al. Hound combines a context-sensitive allocation strategy with an age-segregated memory allocator with the objective of placing related objects of similar age on similar pages. This isolates leaked objects and bloat as the program releases non-leaked objects. Furthermore, Hound performs some virtual compaction to reduce physical fragmentation and therefore reduces the overhead [7].

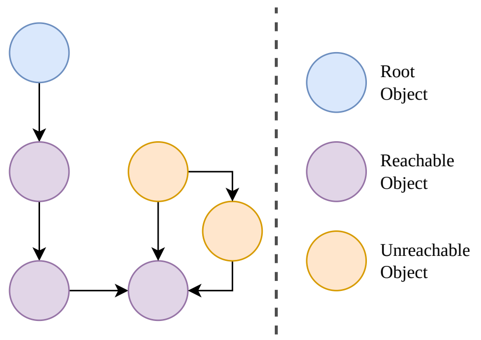

Figure 3: Graphical representation of *Root Objects*, *Unreachable/Leaked Objects* and *Reachable Objects*. Figure adapted from Rahul et al. [9].

4 DETECTION OF MEMORY TUMORS

After elaborating the detection and validation of memory bloat and memory leaks in Section 3, we take a look at unbounded heap growth, which we will also call *memory tumors*.

In contrast to memory cysts, memory tumors are aggregates, which are still used at runtime. The problem with an arising memory tumor is, that data keeps getting added unbounded to an aggregate. As a result, the software crashes at runtime at some point. Moreover, the final heap allocation is not necessarily related to the tumor itself, but just the final blow. Consequently, the debugging of the memory tumor-infected software is cumbersome.

Hill et al. have developed *GrowthTracker*, which aims to detect such memory tumors. To perform a test for memory tumors, the targeted software is run continuously in an automated mode with sample inputs. While this test is running, their tool queries the size of each aggregate periodically. Consequently, a change to the source code is necessary, to track all aggregates in a software. Hill et al. developed a heuristic to identify memory tumors based on the history of the sizes of the aggregates. Furthermore, all aggregates in the target software must be tracked, i. e., when an aggregate is instantiated it must be added to a Central Aggregate Tracker (CAT) and when it is deleted, it must be deleted from the CAT. Moreover, there must be a way for the CAT to query the size of the aggregates [1].

4.1 Aggregate Tracker Injection

In order to achieve the querying of the sizes, Hill et al. decided to facilitate an abstract interface, where each custom aggregate must implement this interface. Since their tool is written for C++ programs, they make use of the available features. This includes multiple inheritance and the use of destructors. The authors utilize two strategies. They define small wrapper classes for the Standard Template Library for C++ and boost, which themselves provide aggregates for developers. In addition, they provide the possibility to create custom aggregates, by only deriving their base tracking class. Every modified aggregate adds itself to the CAT when instantiated and automatically removes itself when destructed [1].

4.2 Tumor Heuristic

Hill et al. have introduced two approaches to detect tumors. In their first naive, growth detection approach, the authors suggested to report statistics every five minutes. In this approach, the tool reports any aggregates that are bigger than in the last iteration to be a tumor. Consequently, the naive approach leads to three unwanted behaviors that should be eliminated [1]. The tool reports

(1) data structures which are oscillating in size.
(2) slowly growing tumors only periodically.
(3) cysts for a long time, although it should not.

For this reason, Hill et al. have developed a more sophisticated approach. They analyze the aggregate history as two intervals. In the first interval, all aggregates are expected to change size very often. Nevertheless, for cysts, the maximum size should stabilize over time. In the second interval, the aggregates show stability by not growing bigger than in the initial interval. Otherwise, this aggregate will be reported as potential tumor.

Interval sizes play a big role in detecting tumors properly. On the one hand, choosing larger interval sizes, minimizes reports of false positives and false negatives. On the other hand, choosing smaller interval sizes, provides immediate feedback to the developer. As a result, the authors of the tool decided to start off with small interval sizes, which are increasing in length exponentially. Thus, immediate feedback is available with low quality, which is increasing over time, because the interval sizes get bigger. The old second interval becomes the first, more accurate interval in the next iteration. Tumors will be detected as soon as the interval is bigger than the growth time of the aggregate.

4.3 Results of GrowthTracker

Hill et al. have detected numerous tumors with their tool GrowthTracker. In their paper, they described their results on four pieces of software, including Orge3D (graphic rendering engine), Bullet (physics engine), WebKit (web browser) and most noticeably Chromium (web browser). The tool could find tumors in most of the programs. Moreover, they found a tumor in the OpenGLSupport library which was used by Bullet. Furthermore, GrowthTracker detected 27 tumors in Chromium by cycling through 1000 popular websites, loading a new site every 20 seconds [1].

5 TUNED MEMORY ALLOCATOR AND REPLACEMENTS

In addition to detecting memory bloat, it is also possible to prevent or reduce its effects by tuning or replacing the memory allocation system. The following section presents a way to tune an existing memory allocator and a replacement that performs page compaction to reduce memory fragmentation.

5.1 Tuned Memory Allocator

It is also possible for memory bloat to occur in odd locations, such as the memory allocator. Lee describes the problem of bloat in allocators that use thread-local caches designed for fast allocation/deallocation on multithreaded systems. The main problem is, that the management operation frequently has to deal with a bloated thread-local cache, resulting in performance degradation.

In his work, Lee proposes a feedback-directed tuning mechanism for the widely used *TCMalloc* created by Google. The optimization technique targets batch sizes, which indicate the amount of objects that are moved during a thread cache management operation, by observing behavior. During a profile run, the system determines optimized values for batch sizes based on a heuristic. The optimized values result in a reduction of management in the internal data structures of TCMalloc. In addition, these optimized values improve the application performance by up to 10% when tested on Google's internal benchmark applications, according to Lee [5].

5.2 Memory Allocator Replacement

Instead of relying on or tuning existing memory allocators, Powers et al. designed Mesh to effectively and efficiently perform memory compaction to reduce memory bloat in the form of fragmentation. Compaction is performed without reallocation, with the use of virtual pages. This is unlike existing heuristic aimed memory allocators that focus on the reduction of the average fragmentation [8].

Mesh utilizes a mechanism called *meshing*, which was first introduced by Novark et al.'s Hound, briefly mentioned in Section 3.2. In Hound, *meshing* is used to reduce the excessive memory consumption caused by the inefficient allocation scheme.

Unlike Hound, Mesh uses meshing in combination with a space-efficient randomized allocation strategy that scatters objects within a virtual page, as shown in Figure 4. This randomized placement on the virtual pages increases the likelihood that two pages will mesh. In addition to the randomized placement, the algorithm also performs a randomized search to increase the chance of finding candidate pages that are likely to mesh. If two candidate pages are found and their objects on the page do not overlap, then these pages can be meshed into a single physical page. The adapted Figure 4 shows the meshing of two virtual pages into a single physical page, with the second physical page being released as indicated by the `munmap` system call.

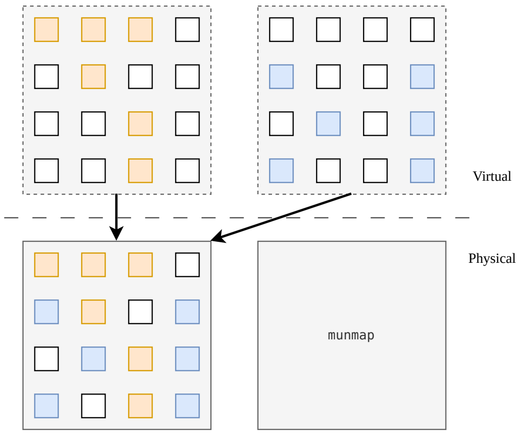

Figure 4: Two virtual pages compacted into a single physical page using Mesh. Figure adapted from Powers et al. [8]

Using Mesh in applications reduces memory consumption by 16% (*Firefox*) to 39% (*Redis*), while still being fast and efficient. The authors plan to explore the integration of Mesh into languages such as Rust and Go, which do not currently support compaction [8].

6 POINTER COMPRESSION FOR DATA STRUCTURES

In contrast to pointer structures, compilers know about the structure of arrays at compile time. Consequently, compilers are much better at analyzing arrays than pointer structures [3].

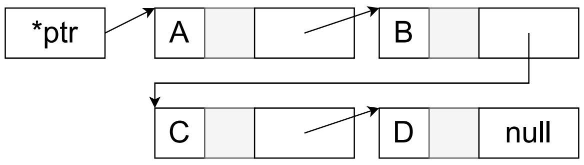

Figure 5: A linked list with 4 byte values stored.

With the goal of reducing memory bloat for pointer structures, Lattner and Adve have introduced the concept of pointer compression for data structures in order to save pointer bloat. Pointer compression implements the idea of reducing the pointer size on 64-bit systems from 8 bytes to 4 bytes. The process of pointer compression is illustrated in Figure 5 and Figure 6, both of which are based on the figures made by Lattner and Adve [4].

This example illustrates a linked list `A->B->C->D`, which lays distributed in memory. Since the nodes may be allocated in any order, we assume, that the order of allocation is `A->C->B->D`.

The resulting pointer compressed data structure can be seen in Figure 6, where the `"pointer"`, which is now an integer, points to the offset 8. There the first element can be found, which is indeed `A`, which has the value 24 following it. This value is again the offset from the pool base to the next value, which points to `B` and so on. In this extreme case, the algorithm achieved a compression of 50% for the data structure.

Further details about pointer compression elaborated in Section 6.3. Pointer compression builds up on data structure analysis [2], explained in Section 6.1 and automatic pool allocation [3], explained in Section 6.2.

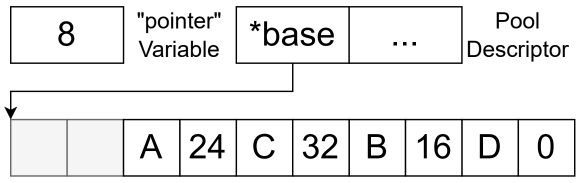

Figure 6: Pointer compressed linked list.

6.1 Data Structure Analysis

Data Structure Analysis (DSA) is a pointer analysis technique that helps identify different instances of data structures in programs. DSA constructs a graph representation of the program. Each node in this graph represents some in-memory objects, and edges between nodes represent pointers [2]. This graph is built using:

- *Context Sensitivity* – The graph representation differentiates objects based on the creation context. Every object has a unique creation path, which is the sequence of function calls leading to the creation of the object. Said path allows DSA to distinguish between instances of data structures.
- *Field Sensitivity* – DSA analyses how each field within a structure is accessed and modified. Consequently, the DSA can accurately model the internal connectivity. This is difficult to support efficiently in non-type-safe languages.
- *Explicit Heap Model* – DSA constructs an explicit heap model of the memory in the program.

DSA is able to analyze large programs quickly with a small memory footprint. Furthermore, DSA has been tested on programs of up to 200,000 lines of code and compile times are roughly 3 seconds. DSA is using up to 5% of the compile time. The result of DSA are the *points-to graph* and *call graph*, which are then used for automatic pool allocation by Lattner and Adve [3].

6.2 Automatic Pool Allocation

Automatic Pool Allocation (APA) is a framework, that segregates distinct data structures on the heap into separate memory pools. Consequently, allowing heuristics to be used to control the internal layout of these data structures on the heap. APA is an intermediate result, in order to achieve pointer compression. In essence, this algorithm operates on programs using calls to `malloc` and `free` and transforms these to use calls to a custom pool library with the functions `poolalloc`, `poolfree` and `poolrealloc` Furthermore, the algorithm uses the graphs resulting from DSA in Section 6.1. An analysis of the graphs determines which descriptors must be available in each function and where they have to be created and destroyed. For each graph-node needing a pool, the algorithm either adds a pool descriptor argument or it allocates a pool descriptor on the stack. Pools that stay within a function are initialized on entry and destroyed at every exit of the function.

The key takeaway is, that automatic pool allocation segregates objects from different data structures into different pools. Furthermore, it guarantees that all variables and fields which are pointing into the heap point to a single pool. Hence, the mapping between pointers and pool descriptors do not have to be tracked dynamically. Thanks to DSA, elements in a pool all have the same type and are aligned in memory, so that the compiler knows the layout relative to the pool base [3] [4].

6.3 Pointer Compression

After talking about DSA and APA, the final step for compressing pointers can be tackled. Figure 5, shows a few nodes, with a size of 16 bytes. Each of the nodes consists of 4 bytes for the value, 4 bytes as padding and 8 bytes for the next-pointer. Moreover, after DSA and APA, the compiler has information about the pool descriptor. This section elaborates the last step, where the data structure from Figure 6 is compressed into the shape of Figure 5. There the values are stored, immediately followed by the byte offset to the pool base of the next node [4].

Pointer compression replaces a 64-bit pointer with an integer index being the byte offset from the start of the pool. Storing offsets instead of absolute addresses leaves room for improvement, because the top bytes are all zeroes. When a pool gets too big, i. e. bigger than 4 GB, they can be expanded again to fit bigger offsets. Pointer compression happens in two steps. First, *index conversion*, where the pointers are replaced by integer values with the offset. Second, *index compression*, trying to use smaller integer types, which might fail, if the pool is bigger than the intended maximum size by choosing a smaller integer type.

7 UTILIZE DATATYPE HEADERS

Traditional approaches to memory management often rely on static analysis, manual optimization, or tracking with extensions to a memory allocator. In this section, we briefly touch on the concept of adding datatype headers to classes and structs as a strategy to provide runtime metadata about stored data types. This is a principle that is already used by various managed languages such as Java and C#.

By including a data type header, developers and analysis tools gain a lot of insight into the data stored within objects, including the identification of pointers. This identification allows for easier traversal of dynamic objects without algorithms such as DSA and thereby identifying sources of bloated constructs. Algorithms may be able to detect bloated constructs more effectively by analyzing memory layouts and relationships between objects.

Despite the potential benefits, there are also a few things to consider when implementing such an approach, with the following being highlighted:

- *Runtime Overhead* – Careful consideration must be given to minimizing runtime overhead while providing sufficient metadata for effective bloat detection and optimization.
- *Complexity* – Managing metadata introduces complexity to the codebase, necessitating careful design and implementation to ensure synchronization between metadata and actual data, particularly in dynamic object traversal scenarios.

Anyway, it would be a considerable idea to provide these data type headers in an optional compiler option for testing purposes, where they are removed for the production environment later on.

8 CONCLUSION

In conclusion, managing memory efficiently in software development, especially in unmanaged languages, is crucial for maintaining performance and stability. As software grows in complexity, so does the risk of memory bloat, which encompasses memory leaks, memory cysts and memory tumors. Detecting and addressing these issues requires a mix of static and dynamic analysis techniques.

Various tools and techniques have been developed to aid in the detection and validation of memory bloat and leaks. Dynamic validation of static leak warnings, runtime data sampling and aggregate tracking are among the methods employed. These approaches enable developers to identify and address memory-related issues efficiently, reducing the risk of performance degradation and crashes.

Furthermore, solutions like GrowthTracker offer ways to detect memory tumors, which are aggregates that grow indefinitely, leading to eventual program crashes. By continuously monitoring aggregate sizes and employing heuristic algorithms, tools like GrowthTracker can pinpoint potential memory tumors, allowing developers to intervene before failures occur.

Additionally, optimizing memory allocators or even replacing them with more efficient systems can mitigate the impact of memory bloat. Techniques, such as feedback-directed tuning for existing memory allocators and innovative approaches like Mesh, show promising results in reducing memory fragmentation bloat and improving application performance.

Moreover, the utilization of datatype headers presents a strategy to provide runtime metadata about stored data types. While this approach may offer insights into data structure and their types, careful considerations must be given to minimize runtime overhead and manage complexity.

In summary, ongoing developments in memory management tools and techniques are essential for ensuring software reliability and efficiency. By adopting the mentioned approaches, developers can proactively address memory-related issues and deliver more robust software solutions.

REFERENCES

[1] Erik Hill, Daniel J. Tracy, and Sheldon Brown. 2013. GrowthTracker: Diagnosing Unbounded Heap Growth in C++ Software. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*. IEEE Computer Society, 134–143. https://doi.org/10.1109/ICST.2013.39

[2] Chris Lattner. 2005. *Macroscopic Data Structure Analysis and Optimization.* Ph. D. Dissertation. University of Illinois Urbana-Champaign, USA. https://hdl.handle.net/2142/81663

[3] Chris Lattner and Vikram S. Adve. 2005. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, Vivek Sarkar and Mary W. Hall (Eds.). ACM, 129–142. https://doi.org/10.1145/1065010.1065027

[4] Chris Lattner and Vikram S. Adve. 2005. Transparent pointer compression for linked data structures. In *Proceedings of the 2005 workshop on Memory System Performance, Chicago, Illinois, USA, June 12, 2005*, Brad Calder and Benjamin G. Zorn (Eds.). ACM, 24–35. https://doi.org/10.1145/1111583.1111587

[5] Sangho Lee. 2017. *Mitigating the performance impact of memory bloat.* Ph. D. Dissertation. Georgia Institute of Technology, Atlanta, GA, USA. https://hdl.handle.net/1853/56174

[6] Mengchen Li, Yuanjun Chen, Linzhang Wang, and Guoqing Xu. 2013. Dynamically validating static memory leak warnings. In *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, Mauro Pezzè and Mark Harman (Eds.). ACM, 112–122. https://doi.org/10.1145/2483760.2483778

[7] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. 2009. Efficiently and precisely locating memory leaks and bloat. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 397–407. https://doi.org/10.1145/1542476.1542521

[8] Bobby Powers, David Tench, Emery D. Berger, and Andrew McGregor. 2019. Mesh: compacting memory management for C/C++ applications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 333–346. https://doi.org/10.1145/3314221.3314582

[9] Jain Rahul, Agrawal Raksha, Gupta Riyanshi, Jain Rajat Kumar, Kapil Neha, and Saxena Ankit. 2020. Detection of Memory Leaks in C/C++. In *2020 IEEE International Students' Conference on Electrical,Electronics and Computer Science (SCEECS)*. 1–6. https://doi.org/10.1109/SCEECS48394.2020.32

[10] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Sevitsky. 2010. Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, Gruia-Catalin Roman and Kevin J. Sullivan (Eds.). ACM, 421–426. https://doi.org/10.1145/1882362.1882448