Automatically Detecting "Excessive Dynamic Memory Allocations" Software Performance Anti-Pattern

Manjula Peiris
Dept. of Computer and Information Science
Indiana University-Purdue University
Indianapolis
Indianapolis, IN, USA
tmpeiris@cs.iupui.edu

James H. Hill
Dept. of Computer and Information Science
Indiana University-Purdue University
Indianapolis
Indianapolis, IN, USA
hillj@cs.iupui.edu

ABSTRACT

This paper presents a methodology for automatically detecting the excessive dynamic memory allocation software performance anti-pattern, which is implemented in a tool named *Excessive Memory Allocation Detector (EMAD)*. To the best of author's knowledge, EMAD is the first attempt to detect excessive dynamic memory allocation anti-pattern without human intervention. EMAD uses dynamic binary instrumentation and exploratory data analysis to determine if an application (or middleware) exhibits excessive dynamic memory allocations. Unlike traditional approaches, EMAD's technique does not rely on source code analysis. Results of applying EMAD to several open-source projects show that EMAD can detect the excessive dynamic memory allocations anti-pattern correctly. The results also show that application performance improves when the detected excessive dynamic memory allocations are resolved.

Keywords

excessive dynamic memory allocation, software performance anti-pattern, dynamic binary instrumentation, detection

1. INTRODUCTION

Dynamic memory allocation [1] is the process of allocating memory "on the fly" at program runtime. In contrast to static memory allocation, programmers do not need to know the exact amount of space or the number of items (*e.g.*, size of an array) at compile time. Dynamic memory allocation operates by using *heap*, or the *free store*, of a program to allocate memory instead of the stack storage of a function. Because the *heap* is global to all scopes of a program (*e.g. classes, functions, and loops*), objects created using dynamic allocations can be shared between different scopes.

Even though dynamic memory allocations provide software developers with memory flexibility at runtime, it is an expensive operation [2]. Allocation and deallocation (*i.e.*, the process of releasing dynamically allocated memory) using standard memory allocation/deallocation functions like malloc/free (in the case of C) and new/delete (in the case of C++) require system calls. Too many dynamic memory allocations can have negative consequences on software performance. For example, Smith et al. [3] detailed how excessive dynamic memory allocations is a software performance anti-pattern. *Software performance anti-patterns* [3, 4], which we just call *anti-patterns* from this point forward, are common designs that have a negative impact on software performance.[1]

Since excessive dynamic memory allocation can negatively impact performance, there are methods to detect it. Unfortunately, the most prominent (and reliable) method for detecting and resolving excessive dynamic memory allocation—and actually any software performance anti-pattern—is (manual) source code analysis [5]. This approach, however, requires expert domain knowledge. More importantly, it requires access to the original source code to support the necessary analysis. As we know, source code may not be readily available when dealing with closed-source applications and/or third-party middleware. Lastly, software performance anti-patterns like excessive dynamic memory allocations are typically visible when a software application is running and placed into a certain state. It is therefore *hard* to identify, evaluate, and resolve the anti-pattern using source code alone.

There are also approaches for detecting software performance anti-patterns without source code. These approaches are either architecture dependent [6] or rule-based [7, 8]. Unfortunately, they do not consider the behavior of software performance anti-patterns at runtime. This makes it *hard* to detect implementation-level anti-patterns like excessive dynamic memory allocations [6].

Solution approach → Analysis supported by dynamic binary instrumentation (DBI). *Dynamic-binary instrumentation (DBI)* [9] is the process of instrumenting a software application at runtime as opposed to recompiling the software application with the instrumentation software. DBI does not require the source code of the system being instrumented because instrumentation logic is injected into the target application while the program's binary is executing. DBI also allows tracing an application and therefore capturing its behavior. DBI therefore allows us to overcome the challenges mentioned above. The remaining challenge now is understanding how to apply DBI to actually detect excessive dynamic memory allocations in an existing application, or middleware. Our proposed technique is based on the intuition that excessive dynamic

ICPE'16, March 12–18, 2016, Delft, Netherlands.

DOI: http://dx.doi.org/10.1145/2851553.2851563

[1]The author Hill experienced this as a visiting researcher at EBay, Inc in 2007. The author was responsible for optimizing the backend search engine for EBay, Inc to address known performance issues. The proposed solution was to remove excessive dynamic memory allocations that were requesting 0 bytes of memory. The solution resulted in 99% improvement in performance for test scenarios that were missing their deadline and 10-15% improvement in performance for scenarios that were not missing their deadline.

memory allocations occur when the software application has many short-lived, high-frequent dynamic memory allocations.

The main contributions of this paper therefore are as follows:

- It presents a method for detecting excessive dynamic memory allocations software performance anti-pattern that has been realized in an open-source tool named *Excessive Memory Allocation Detector (EMAD)*;
- It presents an algorithm to construct a dynamic call graph of a program using an execution trace, which may be missing messages representing routine exit events correspnds to tail calls. This dynamic call graph is used to detect the excessive dynamic memory allocations anti-pattern.
- It showcases how we can apply the K-means clustering algorithm [10] and simple outlier detection techniques to the data collected from DBI to detect excessive dynamic memory allocations anti-pattern;
- It is the first attempt, to the best of the authors knowledge, of a tool that can automatically detect the excessive dynamic memory allocations software performance anti-pattern; and
- It is the first attempt, to the best of the authors knowledge, of using DBI to detect a software performance anti-pattern.

We have applied EMAD to several real open-source projects. Our results from applying EMAD to these open-source projects show that EMAD can report the correct results when the system either exhibits or does not exhibit the excessive dynamic memory allocations anti-pattern. The results also show that when EMAD reports the existence of the anti-pattern, common solutions can be used to resolve the anti-pattern and improve the performance.

Paper organization. The remainder of this paper is organized as follows: Section 2 discusses the challenges associated with detecting excessive dynamic memory allocations anti-pattern. Section 3 discusses the intuition and functionality of EMAD; Section 4 showcases the validity of our approach by applying EMAD to open source software projects. Section 5 discusses related works; and Section 6 provides concluding remarks.

2. MOTIVATION FOR EMAD

Excessive dynamic memory allocation is a common problem that can degrade the software performance. Because of this reason, many software systems and libraries adopt solutions that amortize the cost of allocating/deallocating memory, such as allocating memory from memory pools or free lists [11]. Another solution is to use the Flyweight software design pattern [12]. Although these promising solutions are available, it is *hard* to apply them if one cannot detect the excessive dynamic memory allocation anti-pattern. Unfortunately, detecting the excessive dynamic memory allocation anti-pattern poses several challenges:

1. **Inapplicability of source code analysis techniques.** As mentioned in Section 1, the prominent approach for detecting a software performance anti-pattern is source code analysis. Understanding dynamic memory allocations by just analyzing the source code, however, is *hard*. This is because key information like frequency of object allocation, the size of the object being allocated, and the lifetime of an object are hard to determine at compile time. Moreover, such analysis requires time-consuming code analysis involving experts of complex software systems [13].

   Another limitation of this approach is that it requires source code to be available. Nowadays, most software systems are built using off the shelf software components and libraries. It is therefore ill-conceived to assume that source code is available for analysis at every situation. Even if the source code is available (as with open-source projects), one must still be able to understand the source code (and its intent) in order to search for excessive dynamic memory allocations.

2. **Limitations of software performance anti-pattern detection techniques based on architectural models.** Another approach for detecting software performance anti-patterns is defining rules on performance metric data (*e.g.*, response time and throughput) and/or resource usage data (*e.g.*, CPU and network usage) and then detecting rule violations [6–8]. These rules are defined on architectural models of the system and rule violations are analyzed by simulating the architectural models. Excessive dynamic memory allocation, however, happens at software implementation level. Unfortunately, it is hard to model the implementation details within architectural models [6].

   On the other hand, resource usage data (*e.g.*, high memory footprint) is not a direct indicator of excessive dynamic memory allocations. This is because a function can do a large allocation at once (*e.g.*, a memory pool) and then use it subsequently throughout the entire application lifetime.

3. **Ill-defined excessive dynamic memory allocation problem.** The problem of detecting excessive dynamic memory allocations is ill defined compared to other dynamic memory associated problems like memory leak detection and invalid memory access detection. For example, memory leak detection can be defined as finding dynamic memory allocations that are no longer accessible to the program [14]. Likewise, memory access errors can be defined as detecting invalid reads/writes from/to memory locations.

```
1  struct Foo {
2    int x;
3  };
4
5  int main (int argc, char * argv[]) {
6    Foo * foo = new Foo ();
7    // Do someting with foo
8
9    return 0;
10 }
```

Listing 1: A simple program that has a potential memory leak.

Lisiting 1 illustrates a simple program that has a potential memory leak. As shown in the program, we can conclude that a memory leak exists by examining whether the object *foo* is, or is not, released when the main function returns. Although this examination process can be complex, the problem of detecting the memory leak is well defined.

```
1  struct Foo {
2    int x;
3  };
4
5  int main (int argc, char * argv[]) {
6    for (int i = 0; i < 1000000; i ++) {
7      Foo * foo = new Foo ();
8      // Do something with foo
9      delete foo;
10   }
11   return 0;
```

```
12    }
```

Listing 2: A simple program that has a potential excessive dynamic memory allocation.

> Excessive dynamic memory allocations, however, cannot be defined in such a precise manner. The word "excessive" depends heavily on the context of the allocation. For example, Listing 2 illustrates a simple program that has a potential excessive dynamic memory allocation issue because of the high frequency at which *foo* is being created and deleted. It, however, is hard to determine whether this simple example exhibits excessive dynamic memory allocations by only examining the number of times object *foo* is being created and deleted. This is because excessive dynamic memory allocation is not only based on how many allocations/deallocations occur, but also on the lifetime of those allocated objects.

As discussed above, these challenges make it hard to create automated approaches for detecting excessive dynamic memory allocation anti-pattern. The reminder of this paper will therefore discuss how EMAD helps address these challenges–providing software developers with an improved approach to detect the excessive dynamic memory allocation anti-pattern. This will allow software developers to detect and resolve the anti-pattern problem faster and improve the performance of their software application.

3. THE DESIGN OF EMAD

Figure 1 illustrates EMAD's workflow for detecting the excessive dynamic memory allocations anti-pattern. As shown in the figure, the process consists of 3 major steps: (1) instrumenting the software application using DBI to collect an execution trace; (2) constructing a call graph of the software from the collected execution trace; and (3) analyzing the call graph to detect excessive dynamic memory allocations. We discuss each step in detail throughout the remainder of this section.

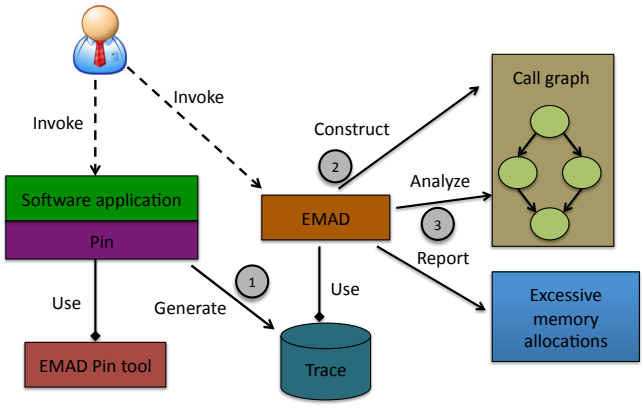

Figure 1: Conceptual overview of EMAD's workflow.

3.1 Instrumenting the Software Application

EMAD uses Pin [9] along with Pin++ [15] as the underlying DBI framework to instrument an application and collect the needed execution trace. Sidebar 1 provides a brief overview on Pin and Pin++. EMAD uses Pin++ to implement a Pintool that instruments a program at routine level. The Pintool instruments each routine call at start and at exit. The Pintool then generates an execution trace where each message in the execution trace contains the following information:

- **Thread id**. The thread id is a unique identifier of the thread calling the routine under analysis. This is important because the caller-callee relationships between routines is determined on a per thread basis when constructing the call graph. The thread id therefore is used to uniquely identify the thread.
- **Routine id.** The routine id is a unique id of the routine assigned by Pin. This information is important because the routine name is not unique if the same routine is in different image or if it is overloaded in the same class. This allows EMAD to uniquely identify each routine it instruments.
- **Event name.** The event name represents the type of event that is occurring. For EMAD, the event name is either *start* or *exit*. Start represents the beginning of a routine call and exit represents the return of a routine call. This information is important because it determines what subprocedures (*i.e.*, the sub-procedure for receiving a start event or the sub-procedure for receiving an exit event) to call in Algorithm 1.
- **Name.** The name represents the undecorated name of the routine under instrumentation (or being analyzed). This piece of information is important because this allows EMAD to report the human readable name of a routine when it identifies the location(s) of excessive dynamic memory allocations.

Sidebar 1: Pin and Pin++

Pin is a DBI tool for IA-32 and X86-64 instruction-set architecture. Pin provides a framework to implement analysis tools called Pintools. Pintools analyze different aspects of a program, such as program faults, program behavior, root causes, and performance profiling. Pintools can also analyze a program at different levels of granularity: binary image level, routine level, and instruction level.

Even though Pin provides several facilities to instrument programs, the Pintools implemented using Pin are fragile, rigid, hard to extend/reuse, and difficult to understand [15]. Pin++ provides an object-oriented, template meta programming approach to writing Pintools that handle the above mentioned software engineering issues. Moreover, Pintools implemented using Pin++ have a reduction in cyclomatic complexity, do not induce additional overhead, and improves the Pintools performance in certain cases. For example, Hill et al. [15] has shown that Pin++ can have a 54% reduction in complexity, increase modularity, and up to 60% reduction in instrumentation overhead when compared to traditional Pintools.

Because EMAD eventually constructs a call graph (see Section 3.2) that records dynamic memory allocations and deallocations, EMAD assumes signatures with the patterns shown in Listing 3 for dynamic memory allocation and deallocation routines. The patterns in this listing are the common signatures for most of the general-purpose memory allocation/deallocation routines in both standard libraries (*e.g.*, malloc/free and new/delete) and third-party libraries that implement custom memory management.

```
1  // Pattern expected for memory allocation routine.
2  void * [allocation_method] (size_t size);
3
4  // Pattern expected for memory deallocation routine.
5  void [dellocation_method] (void * location);
```

Listing 3: Allocation/Deallocation method signatures.

EMAD also collects the following additional details for allocation/deallocation routines in the execution trace:

- **Allocation size.** This is the input parameter at the start of the allocation routine, which is the size of the allocation. It is used to characterize the memory allocation.
- **Address of the allocation.** This is the return value at the exit of the allocation routine, which is the allocated memory location address. It is used to correlate memory allocations and deallocations.
- **Allocation timestamp.** This is the exiting timestamp from the allocation routine, and specifies the time when the memory allocation was active. It is used to calculate the lifetime of the corresponding memory allocation.
- **Deallocation timestamp.** This is the exiting timestamp from the deallocation routine, and specifies the time when the memory allocation was deactivated. It is used to calculate the lifetime of the corresponding memory allocation.

The execution trace (*i.e.*, the data discussed above) is recorded by the Pintool while the program under instrumentation is executing. Listing 4 shows a portion of an example execution trace the EMAD Pintool will generate. Once the execution trace is recorded, the remainder of EMAD's analysis is done offline.

```
 1   0 19 start main
 2   0 20 start Initialize
 3   0 22 start malloc 32
 4   0 22 exit malloc 842c008 141677579
 5   0 20 exit Initialize
 6   0 34 start operation1
 7   0 22 start malloc 64
 8   0 22 exit malloc 9786cd0 14167757886
 9   0 35 start operation2
10   0 23 start free 9786cd0
11   0 23 exit free 14167757928
12   0 23 start free 842c008
13   0 23 exit free 14167757928
14   0 35 exit operation2
15   0 34 exit operation1
16   0 19 exit main
```

Listing 4: Example execution trace generated by EMAD

3.2 Constructing the Call Graph

EMAD uses the execution trace collected during the instrumentation step (see Section 3.1) to construct a call graph [16] of the program. The constructed call graph is a weighted directed graph. Each node in the graph represents an executed routine in the application. Each edge represents a caller-callee relationship. The edge weights represent the frequency of each routine call. The Figure 2 illustrate the call graph EMAD will be constructing for the execution trace shown in Listing 4.

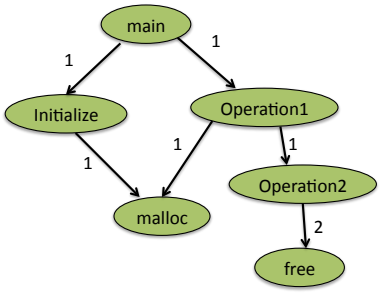

Figure 2: Call Graph for the execution trace in Listing 4.

Algorithm 1 General algorithm for constructing the call graph

```
1:  procedure ConstructCallGraph(ET)
2:      ET : set of routine start/exit messages from execution trace
3:
4:      CG : Call graph
5:      CS : Set of stacks of called routines, one per each thread
6:
7:      for all ET_i ∈ ET do
8:          j ← extract_thread_id(ET_i)
9:          R ← extract_routine(ET_i)
10:         if ET_i is a routine start trace then
11:             HandleRoutineStartTrace(CG, CS_j, R)
12:         else if ET_i is a routine exit trace then
13:             HandleRoutineExitTrace(CG, CS_j, R)
14:         end if
15:     end for
16:
17:     for all k ∈ thread ids do
18:         while CS_k is not empty do
19:             R ← Top(CS_k)
20:             HandleRoutineExitTrace(CG, CS_k, R)
21:         end while
22:     end for
23:
24: end procedure
```

The constructed call graph is also a condensed graph [17]. This is because EMAD is not representing every call to a routine as its own node and edge as in a detailed call graph. Instead, EMAD is capturing how many times a routine is called. The condensed call graph reduces the amount of resources needed to construct the needed call graph of an application. More importantly, we have learned that a detailed call graph makes it hard to perform the necessary analysis to detect excessive dynamic memory allocations.

Algorithm 1 details EMAD's process for constructing the call graph from an execution trace. The algorithm consists of two sub-procedures. The first sub-procedure handles routine start messages (line 11). The second sub-procedure handles routine exit messages (line 13). It is worth noting that Algorithm 1 maintains a *called routine stack* for each thread in the application being instrumented. This is because caller-callee relationships are maintained on a per thread basis when using the condensed graph approach [17]. There, however, will be one call graph that is updated using the relationships maintained in each call stack.

The sub-procedure for handling routine start messages is shown in Algorithm 2. Whenever a routine start message is found, the corresponding routine object is pushed onto the stack. A node representing the routine object is also added into the call graph. Because EMAD is constructing a condensed call graph, the *AddNode* statement (line 7) only adds a node to the call graph if and only if the node is not in the call graph.

Algorithm 2 Procedure that handles a routine start trace.

```
1:  procedure HandleRoutineStartTrace(CG, cs, R)
2:      CG : Call graph
3:      cs : The routine stack of a thread
4:      R : The routine
5:
6:      Push(cs, R)
7:      AddNode(CG, R)
8:  end procedure
```

The sub-procedure for handling routine exit messages is not as straightforward when compared to the sub-procedure for handling routine start messages. This is because the instrumentation of routine exits does not work reliably in the presence of tail calls or when return instructions cannot reliably be detected under Pin [18]. From our experience, a majority of the routine exit messages for the corresponding routine start messages can be found in the execution trace. When a routine exit message cannot be found in the execution trace, EMAD uses Algorithm 3 to resolve the *missing exit message* problem.

Algorithm 3 Procedure that handles a routine exit trace.

```
 1: procedure HANDLEROUTINEEXITTRACE(CG, cs, R)
 2:     CG : Call graph
 3:     cs : The routine stack of a thread
 4:     R : The routine
 5:
 6:     if cs is not empty then
 7:         if Top(cs) = R then
 8:             Pop(cs)
 9:             if cs is not empty then
10:                 AddEdge(CG, Top(cs), R)
11:             end if
12:         else
13:             while Top(cs) ≠ R do
14:                 r ← Top(cs)
15:                 Pop(cs)
16:                 if cs is not empty then
17:                     AddEdge(CG, Top(cs), r)
18:                 end if
19:             end while
20:
21:             Pop(cs)
22:             if cs is not empty then
23:                 AddEdge(CG, Top(cs), R)
24:             end if
25:         end if
26:     end if
27:
28: end procedure
```

As shown in this algorithm, it first checks whether the routine object at the stack top is the same as the routine object represented from the message. If this condition holds true, then this implies that the routine object has both start and exit messages in the execution trace. It also implies that the caller of the routine should be the stack top element once the current stack top is removed. EMAD therefore creates an edge between the two routines with the correct directionality (line 7-10) if an edge does not already exist. If an edge already exists, its weight is increased by 1. The *AddEdge* (line 17) implements this logic.

When the routine object at the top of the stack and the routine object correspond to routine exit message mismatches, it implies that the routine exit message for the routine object at the top of the stack is missing. The allocation object's caller should be current stack top's adjacent routine object. EMAD therefore saves the stack top, pops an element from the stack, and connects the new stack top with the previous stack top. EMAD continues this process until it finds the routine object represented by the current routine exit message. The sub-procedure for handling routine exit messages therefore guarantees that the correct caller-callee relationship is preserved even when routine exit messages are missing in the execution trace.

Once all messages in the execution trace are processed, there can still be routine objects remaining on the stack. EMAD explicitly calls the *HandleRoutineExitTrace* routine (line 20 of Algorithm 1) while iterating through call stacks of each thread. This is necessary because the routine exit messages of the remaining routine objects is missing. Explicitly calling *HandleRoutineExitTrace* will complete the call graph with any missing edges.

As mentioned in Section 3.1, the start/exit messages for allocation/deallocation routines contain extra details such as parameter/return values and timestamps. Algorithm 1 and its sub-procedures discussed above will extract and store this additional information in allocation/deallocation routine objects during the execution trace processing. The data associated with the allocation/deallocation routines is used to create *allocation objects*. An allocation object has three attributes, the size of the dynamic memory allocation; the routine that calls the memory allocation routine to allocate memory; and the routine that calls the memory deallocation routine. In EMAD, each dynamic memory allocation during the lifetime of the application is represented using an allocation object.

An allocation object is distinguishable from another allocation object if any of its attributes is different. One would think it should be possible to use the address of the memory allocation to uniquely represent an allocation object. This, however, is not possible because the same memory address can be reallocated several times during the lifetime of the application. The memory address of an allocation is therefore not unique once we consider the entire lifetime of the application. EMAD therefore only uses the memory address of an allocation to match the caller of the allocation routine and caller of the deallocation routine.

Each allocation object also has a frequency. The frequency specifies how many times an allocation object (with same values for above three attributes) occurs throughout the software application lifetime. For each allocation object, we can also calculate its lifetime as follows:

$$T_l = T_d - T_a \tag{1}$$

where T_l represents the lifetime of the allocation object; T_d represents the timestamp of the deallocation exit message; and T_a represents the timestamp of the allocation exit message. Each distinct allocation object stores its average lifetime. Lastly, EMAD uses the two attributes of an allocation object, *i.e.* its frequency, its calculated average lifetime, and the constructed call graph to detect the excessive dynamic memory allocation anti-pattern.

3.3 Detecting Excessive Dynamic Memory Allocations

As mentioned in Section 1 our analysis technique for detecting excessive dynamic memory allocations is based on the intuition that this anti-pattern occurs when the software application has many short-lived high-frequent allocation objects. Our intuition comes from studying the two main solutions used to resolve the excessive dynamic memory allocation software performance anti-pattern [3].

The most common solution to resolve this anti-pattern is to use a custom memory allocator [19]. The basic idea of a custom memory allocator is to use a memory pool. When using a memory pool, a large chunk of memory is allocated during the software application initialization phase. The subsequent requirements for memory allocations are fulfilled by obtaining memory from this memory pool—thereby eliminating the system calls to allocate memory. When the allocated memory is no longer needed, it is released into the memory pool—thereby eliminating the system calls to deallocate memory.

The custom memory allocations approach will not be effective

if the allocation objects are in use for long periods of time. This is because when there are many such objects, eventually the memory pool will not be able to fulfill the allocation requests. This will result in acquiring memory from the operating system and the expected performance gain may not be achieved. When the software application has high-frequent short-lived allocation objects, however, the memory pool regains the memory it has given to the application. This improves the performance by rarely allocating memory using general purpose memory allocators.

The other solution for the excessive dynamic memory allocation anti-pattern is to use the Flyweight software design pattern [20]. The Flyweight software design pattern is similar to using a custom memory allocator. Its strategy is also based on reusing the already allocated objects. The only difference is the Flyweight design pattern applies the solution at a higher level of abstraction such as reusing particular types of objects. It is also effective only when there are high-frequent short-lived object instances that are reusable.

Based on this intuition, EMAD's main goal in the detection process is to identify short-lived, high-frequent allocation objects. EMAD analyzes the frequency and average lifetime of the allocations objects annotated with the allocation/deallocation routines in the constructed call graph. To understand the analysis process, we introduce a frequency-lifetime diagram as illustrated in Figure 3. Each point in the diagram represents a unique allocation object. The x value represents the frequency of the allocation and y value represents the average lifetime of the allocation. We consider points that fall in the low-right quadrant to correspond to short-lived, high-frequent dynamic memory allocations. These are the set of points we want to identify in our analysis.

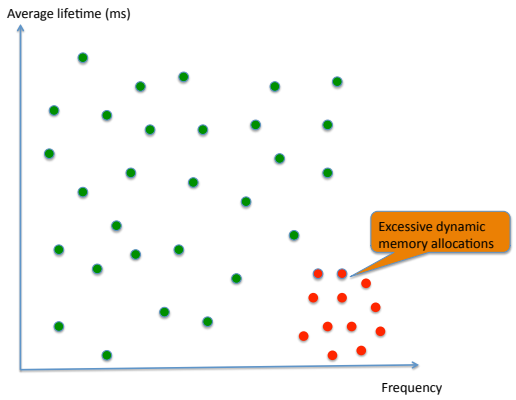

Figure 3: Frequency-lifetime diagram.

Because frequency and lifetime of allocation objects are dependent on each software application, it is hard to define thresholds to filter high-frequent, short-lived memory allocations. EMAD therefore provides two different exploratory data analysis techniques: one using K-means clustering, and the other using an outlier detection technique to identify high-frequent, short-lived memory allocations.

3.3.1 *Using K-means clustering*

Clustering is a non-supervised technique that can be used to partition objects based on the quantitative values of their attributes. The goal of clustering is to partition regions of points that have similarities. To accomplish this task, EMAD uses popular K-means algorithm [10] to cluster the allocation objects based on their frequency and average lifetime.

Once the allocated objects are clustered, EMAD then checks whether there is a cluster C that satisfies all the following conditions:

1. The average frequency of C's members is the highest compared to the other clusters. This information is important because if the frequency is high, then there is a potential excessive dynamic memory allocation issue.

2. The average lifetime of C's members is the lowest compared to the other clusters. This information is important because when the allocation object is a short lived object there is a potential excessive dynamic memory allocation issue.

If EMAD can find a cluster that satisfies both the conditions above, then it reports that software application has excessive dynamic memory allocation anti-pattern. The report may contain all the members of that cluster, or a user-defined number n of members. In the latter case, EMAD will report first n members in the descending order of frequency. Because the allocation objects contains the caller information of the allocation, EMAD can also report call hierarchy of the allocation similar to other dynamic memory analysis tools (*e.g.*, Valgrind [14]). By providing the call hierarchy software developers can quickly locate the excessive dynamic memory allocations anti-pattern in the source code—eliminating tedious and time consuming source code analysis.

On the other hand, if EMAD cannot find a cluster that satisfies the conditions above, then EMAD reports that the software application does not have the excessive dynamic memory allocations anti-pattern. This is because the partitioning indicates that most of the high-frequent allocation objects have a longer lifetime, or short lived allocation objects are not frequent.

Because EMAD's analysis is based on a clustering technique, the user has to configure the parameter that controls the number of clusters. This parameter, in turn, controls the number of partitions EMAD has to create from the dataset. Unfortunately, this is one of the limitations in cluster analysis [21, 22]. Likewise, identifying the correct number of clusters may require some trial and error.

3.3.2 *Using outlier detection*

In this technique, we convert the two dimensional dataset into a one dimensional dataset by calculating the ratio between frequency and average lifetime of each allocation object.

Therefore, the ratio R is defined as:

$$R = \frac{frequency}{lifetime} \tag{2}$$

According to the above equation the value of R is larger when the frequency is high and lifetime is low. Therefore we consider allocation objects that have relatively high values as potential excessive dynamic memory allocations. Based on this intuition we consider extreme outliers of this one dimensional dataset as potential excessive dynamic memory allocations. We only consider positive outliers that have larger values for R, not the outliers with lower values. To identify these extreme values we use Interquartile Range (IQR) based outlier detection technique [23]. We adopt this technique instead of standard score based outlier detection techniques because we observed that the data in our datasets are not normal distributions [24]. We consider allocation objects that have a value greater than the value obtained from the following expression as potential dynamic excessive memory allocations.

$$Q_3 + \mu \times IQR \tag{3}$$

Here Q_3 is the third quartile, IQR is the Interquartile Range, and μ is a user provided parameter. If we increase the value of μ,

EMAD may miss potential excessive dynamic memory allocations; and a lower value for μ may cause EMAD to report several false-positives. Therefore, the user has to provide a reasonable value for μ which may requires some trial and error. A good initial value for μ is the value obtained for IQR. Another way to decide on a value for μ is to first view the datasets and see how the value of R is deviating from normal. EMAD outputs this value during the analysis. EMAD also ranks the excessive dynamic memory allocations based on the value of R. Therefore, users can get an idea about the relative significance of excessive dynamic memory allocations after seeing the results. EMAD also provide facilities to view both two dimensional (*i.e.* frequency and lifetime) and one dimensional datasets (*i.e.*, value of R) of allocation objects.

4. EVALUATION OF EMAD

This section illustrate how we validate EMAD's methodology by applying it to several real world open source systems. Validating EMAD's technique is challenging, because once EMAD reports excessive dynamic memory allocations we need to make sure it is an actual excessive dynamic memory allocation, which has an impact on system performance. We therefore validated EMAD with following types of experiments: (1) known released software version that has the anti-pattern and then a newer version of the same software without the anti-pattern; (2) software that have the anti-pattern, which is previously unknown; (3) an anti-pattern induced software version to see whether EMAD can detect the induced anti-pattern; and (4) software that does not have the anti-pattern to see if EMAD does not identify any problems. Lastly, we evaluated performance before and after resolving the anti-pattern for all experimental scenarios.

4.1 Experimental Setup

We used the following open-source projects in our experiments:

1. **SQLite** (www.sqlite.org) is a SQL database engine primarily used in embedded devices, such as mobile phones and web browsers. We selected SQLite for our experiments because we searched its release history and identified versions that were impacted by the excessive dynamic memory allocation anti-pattern. This project will evaluate if EMAD is able to identify the routine that is the source of the problem.

2. **TAO** (www.cs.wustl.edu/~schmidt/TAO) is an implementation of the CORBA specification used in distributed real-time and embedded systems. We selected TAO because its application domain values small percentages in performance improvements. Also the excessive dynamic memory allocation was not reported in TAO before applying EMAD.

3. **Axis2-C** (axis.apache.org/axis2/c/core) is a web services framework that is implemented in C using the popular Axis2 SOAP processing architecture. Axis2-C is used in some of the modern cloud computing infrastructure middleware and also in scripting language based web services engines [25]. We selected Axis2-C because we could induce the dynamic memory allocations anti-pattern. This will evaluate if EMAD can detect the induced anti-pattern.

4. **Xerces-C++** (xerces.apache.org/xerces-c) is a C++ framework for manipulating XML documents. We selected Xerces-C++ because it allows developers to integrate custom memory allocators to improve performance.

All experiments were conducted on an Intel core 2 Duo 3.33 GHz processor, with 4GB memory and running 32-bit Ubuntu 14.04 operating system. We also used Pin 2.13 and Pin++ 1.0.0-beta.

4.2 Experimental Results for SQLite

We used the Northwind database [26] for our SQLite experiments. We used a single SQL file that contained SQL statements for table creation, data insertion, table updating, and data querying. The SQLite command line interface was used to interpret the SQL file. Lastly, performance was measured by recording total time to process the Northwind database SQL file.

According to the SQLite [27] release history, SQLite had the excessive dynamic memory allocations software performance anti-pattern prior to version 3.6.1. Such versions created many number of short-lived memory allocations in each database connection. The SQLite documentation states the following related to this excessive dynamic memory allocations problem[2]:

> These small memory allocations are used to hold things such as the names of tables and columns, parse tree nodes, individual query results values, and B-Tree cursor objects. There are consequently many calls to *malloc* and *free*—so many calls that *malloc* and *free* end up using a significant fraction of the CPU time assigned to SQLite.

As a solution to this issue, SQLite developers implemented a custom memory allocator called *lookaside allocator* that preallocates a large chunk of memory and divides it to fixed size small slots inside each database connection. We therefore applied EMAD against SQLite 3.5.9. We did not use SQLite 3.6.0 because it was not a stable release.

4.2.1 Experimental results using clustering

In our experiments, EMAD detected 3 locations where SQLite 3.5.9 was performing excessive dynamic memory allocations. The 3 locations are shown in Table 1. EMAD also generated the call-tree for routines in Table 1. For example, Listing 5 illustrates the call-tree for the *sqlite3DbMallocRaw* routine. The call-tree shows the routine name and frequency (inside parentheses) of each caller-callee relationship. Although there are several call-trees for the *sqlite3DbMallocRaw* routine,Listing 5 only shows the call frequencies with maximum edge weights. Due to space limitations we discarded the other call-trees for *sqlite3DbMallocRaw* routine.

```
1  sqlite3_column_name (10659)
2  sqlite3_step (14098)
3  sqlite3VdbeExec (60800)
4  sqlite3BtreeNext (378600)
5  sqlite3VdbeMemRelease (788186)
6  sqlite3_prepare (3450)
7  sqlite3LockAndPrepare (3450)
8  sqlite3Prepare (3450)
9  sqlite3RunParser (102322)
10 sqlite3Parser (26436)
11 sqlite3Expr (26765)
12 sqlite3DbMallocRaw (45858)
```

Listing 5: Partial Call-tree for the routine *sqlite3DbMallocRaw*.

As described in the SQLite documentation, routines like *sqlite3_step* and *sqlite3_column_name* contribute to excessive dynamic memory allocations in SQLite3. As shown in Listing 5, EMAD is able to report these routines in the call-tree for *sqlite3DbMallocRaw* routine as a source of the excessive dynamic memory allocations.

Figure 4 shows the frequency-lifetime diagram for this experiment, which supports the reported excessive dynamic memory allocations. As shown in Figure 4, the 3 allocation objects that correspond to excessive dynamic memory allocations have high-frequency

[2]More on the quote can be found at the following location: www.sqlite.org/malloc.html#lookaside

Table 1: Excessive dynamic memory allocation locations in SQLite-3.5.9 identified by clustering method

Caller	Size	Destroyer	Freq.	Avg. Lifetime
sqlite3DbMallocRaw	68	sqlite3ExprDelete	29394	929.758
sqlite3DbMallocRaw	32	sqlite3VdbeMemRelease	12918	224.889
pager_write_size	1024	sqlite3BtreeCommitPhaseTwo	6832	1.29202

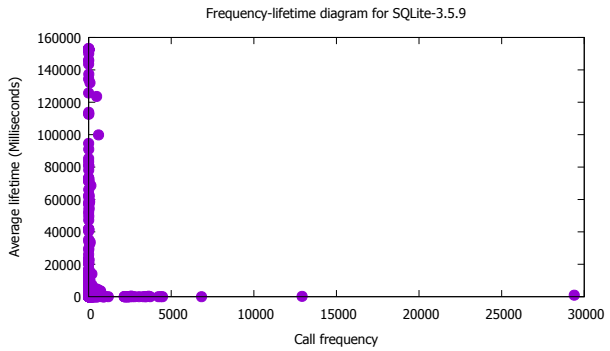

Figure 4: Frequency-lifetime diagram for SQLite-3.5.9

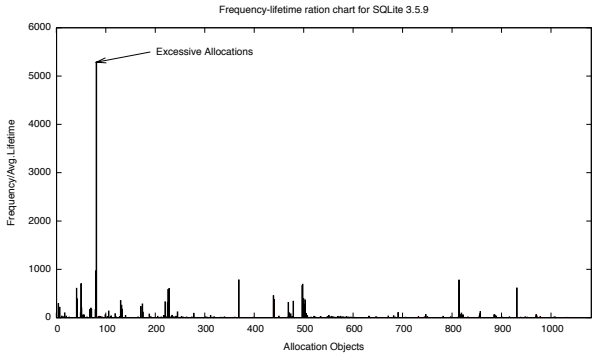

Figure 6: Frequency-lifetime ratio chart for SQLite-3.5.9

(as high as 29394) and short lifetime (as low as 1.29202ms) when compared to the other allocation objects in the figure.

SQLite releases after version SQLite 3.5.9 implement the solution to the excessive dynamic memory allocations anti-pattern. To verify this, we applied EMAD to SQLite 3.8.5. In this version, EMAD identified *memjrnlWrite* as the only location to perform excessive dynamic memory allocations. This location is related to an I/O operation that has no relation with the excessive dynamic memory allocation problem we found in SQLite 3.5.9. The frequency-lifetime digram shown in Figure 5 validates the results of EMAD. As shown in the diagram, there is only one allocation object that resides in high-frequency, short-lifetime region.

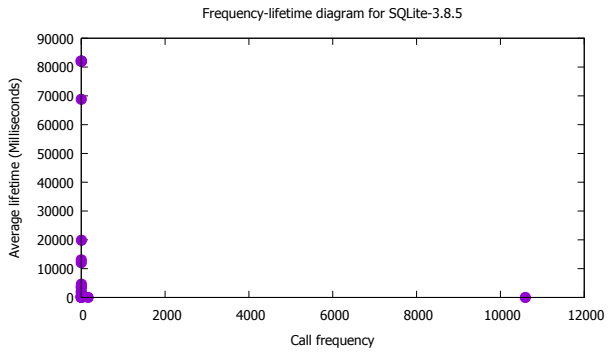

Figure 5: Frequency-lifetime diagram for SQLite-3.8.5

4.2.2 *Experimental results using outlier detection*

We applied EMAD to SQLite 3.5.9 after configuring EMAD to use its outlier detection technique. After using a value of 1000 for μ in Equation 3, EMAD reported *pager_write* as the only location with excessive dynamic memory allocations as shown in Figure 6.

The outlier detection technique did not categorize some of the high frequency, short-lifetime allocation objects as excessive dynamic memory allocations. This is because the IQR of the dataset is as low as 1.7211 and we had to use a value as larger as 1000 for μ to filter the outliers. Unfortunately, a lower μ value started producing false positives. For example, when we lowered the value of μ, EMAD reported allocation objects that have a frequency of 162 and an average lifetime of 0.2075 msec as excessive dynamic memory allocations. Although the average lifetime of the allocation objects is low, the frequency is also low when compared to frequencies of excessive dynamic memory allocations. Lastly, we applied the outlier detection technique to SQLite 3.8.5. EMAD reported the same location shown in Figure 5 from the clustering technique.

4.2.3 *Resolution and performance improvements*

To resolve the identified problem, we used a custom memory allocator (as mentioned in the SQLite documentation) to resolve the performance anti-pattern and improve the performance. According to SQLite documentation, the custom memory allocator preallocates a chunk of memory during application initialization. To apply to solution, we re-compiled SQLite-3.8.5 with the custom memory allocator enabled. We then ran the same experiment with the enabled custom memory allocator. For our experiments, the custom memory allocator improved performance by 10%.

Table 2: Performance of different versions of SQLite

SQLite Version	Total Process Time	# of mallocs
3.5.9	475.01 ms	184859
3.8.5	338.43 ms	58441
3.8.5 w. custom allocator	308.53 ms	9706

To summarize our performance results, Table 2 shows the total processing time of the Northwind database SQL file for each version SQLite we used in our experiments. As shown in the table, the performance of SQLite improved after we applied each solution to the identifed excessive dynamic memory allocation software performance anti-pattern. For example, SQLite 3.8.5 improved approximately 30% in performance when compared to SQLite 3.5.9. Likewise, SQLite 3.8.5 with custom memory allocator improved approximately 10% when compared to SQLite 3.8.5 without the custom memory allocation. More importantly, the experiments show

EMAD was able to detect the excessive dynamic memory allocations and can assist developers in improving performance.

Lastly, Table 2 also shows the number of malloc/free routine calls invoked by each version of SQLite we used in our experiments. We collected this data using a Pintool that counts malloc/free routine calls. Our results show that when the excessive dynamic allocation anti-pattern is resolved, there are fewer system calls to malloc/free.

4.3 Experimental Results for TAO

We applied EMAD to TAO while sending 10,000 requests to its echo service example. EMAD reported two locations with excessive dynamic memory allocations[3]: (1) (*CORBA::string_alloc*, *CORBA::string_free*) and (2) (*operator »*, *IOP::ServiceContextList:: ServiceContextList*). For this experiment, both the clustering and outlier detection technique reported the same locations.

The first excessive dynamic memory allocation is coming from TAO. The second one is coming from the echo service (*i.e.,* the application) when it is echoing the received string. The frequency-lifetime diagram in Figure 7 and the frequency-lifetime ratio chart in Figure 8 confirm EMAD's findings. Apart from the two excessive dynamic memory allocations, almost all the other allocation objects have a very low frequency. Because of this, only the two data points that correspond to the excessive dynamic memory allocations are visible in the Figure 8.

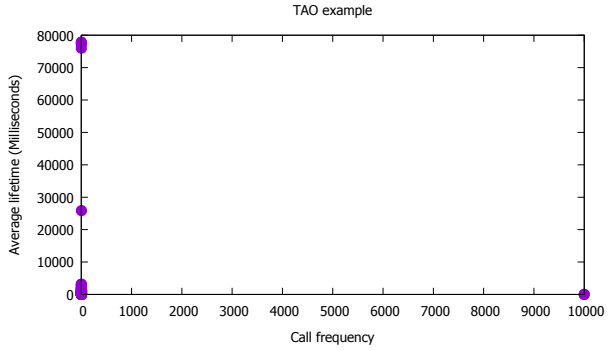

Figure 7: Frequency-lifetime diagram for TAO

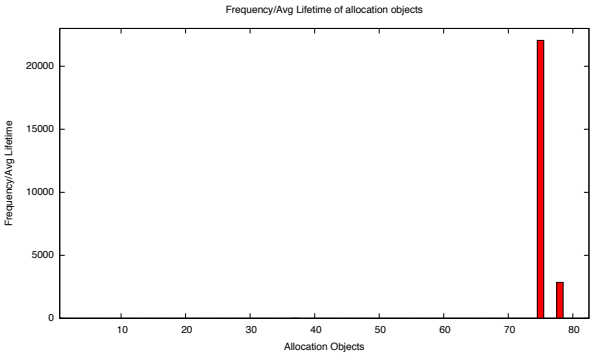

Figure 8: Frequency-lifetime ratio chart for TAO

Our focus was on resolving the identified problem that resided in TAO because it will impact all the applications that use TAO.

[3]We are only listing the caller and destroyer location due to space limitations

This excessive dynamic memory allocation occurs when TAO performs a zero size allocation using *operator new* to allocate a list of buffers for service context information. When the same client sends many requests, however, the buffer can be allocated only for the first request. Our simple fix was to return immediately before calling *operator new* when the requested length is 0.

After this fix we re-evaluated TAO's performance, and measured the time it takes to process n requests. We observed a 5-10% performance gain for larger number of requests. The performance results are shown in the Table 3.

Table 3: Performance of echo service example in TAO.

# of Requests	Before Fix (sec)	After Fix (sec)	Gain
10K	2.275431	2.25299	0.98%
20K	4.589058	4.491926	2.11%
30K	6.972080	6.825455	2.1%
40K	9.51474	9.419871	0.99%
50K	11.487203	11.291216	1.7%
100K	22.917998	22.587449	1.44%
200K	52.195151	45.445869	12.93%
300K	68.968680	63.624066	7.74%
400K	91.914805	85.586583	6.88%
500K	115.174436	106.963704	7.12%

We reported our findings to the TAO mailing list. The TAO developers accepted the patch as it was something they were not aware of. Although it is not a bug, they were willing to fix the problem because even a small improvement in performance is valuable in the context of distributed realtime and embedded systems.

4.4 Experimental Results for Axis2-C

Axis2-C uses Apache's memory pool routines to dynamically allocate memory. To induce the excessive dynamic memory allocations anti-pattern, we changed the Axis2-C module to use malloc/free functions. After applying the change, we used Apache Benchmark tool to send 2,000 SOAP requests to Axis2-C sample echo service deployed in an Apache Web Server, and instrumented the Apache Web Server with Axis2-C while the requests were processed. Finally, the collected execution trace was analyzed by EMAD for excessive dynamic memory allocations.

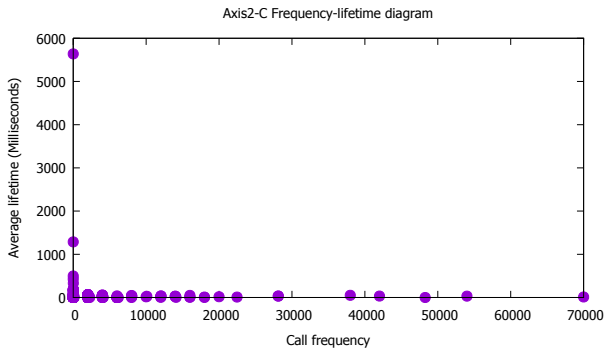

Figure 9: Frequency-lifetime diagram for Axis2-C

As shown in the frequency-lifetime (see Figure 9) and frequency-lifetime ratio (see Figure 10) diagrams, we found several locations where Axis2-C performs excessive dynamic memory allocations. The first five locations based on rank for the clustering technique was *axutil_string_create*, *axiom_node_create*, *axutil_hash_first*, *axutil_hash_find_entry*, and *axutil_string_create_assume_ownership*.

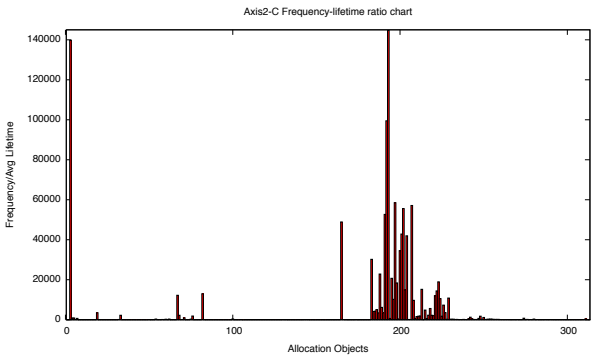

Figure 10: Frequency-lifetime ratio chart for Axis2-C

Likewise, the first five locations based on rank for the outlier technique was *guththila_get_prefix*, *axutil_hash_first*, *guththila_get_prefix*, *axutil_strdup*, and *axutil_stracat*.

From our analysis, Axis2-C's excessive dynamic memory allocations happens mainly because of deep string copies. When used with Apache Web Server, Axis2-C can still perform deep copying when necessary without sacrificing performance by leveraging Apache's memory pools. When using Apache memory pools, Axis2-C has 8% of performance improvement for processing 1 million requests as shown in Table 4. The table also shows there are 96% fewer calls to malloc when processing a single request.

Table 4: Axis2-C performance.

Item	w. memory pools	w.o. memory pools
1 million requests	280 secs	304 secs
Mallocs/request	370	11032

4.5 Experimental Results for Xerces-C++

We used Xerces-C++ Simple API for XML (SAX) command-line utility to parse a 117 KB XML file that contained 1,318 elements and 71,166 characters. We then used EMAD to collect the execution trace of the SAX command-line utility while it processed the XML file. Next, we used EMAD to generate the call graph from the execution trace and detect the presence of the excessive dynamic memory allocation software performance anti-pattern.

In this experiment, EMAD could not find any excessive dynamic memory allocations using the clustering or outlier detection technique. We also checked if previous versions of Xerces-C++ had the excessive dynamic memory allocation software performance anti-pattern. We, however, could not find any version reviewing Xerces-C++ release history.

Since Xerces-C++ supports custom memory allocators, we investigated whether we could improve Xerces-C++ performance by implementing a custom memory allocator. By default, Xerces-C++ uses the new/delete operators to allocate/deallocate memory. Our custom memory allocator is an implementation that uses a free list. At the beginning, it allocates a large chunk of memory that is partitioned into small user defined chunks. The small chunks are maintained as two linked list. The first linked list maintains the memory chunks that are being used in the program. The second linked list maintains the freely available memory chunks.

The allocation function returns a memory chunk from the free list and creates a pointer to that chunk from allocated list. The deallocation function gives back the deallocated memory chunk to the free list and removes the corresponding pointer from the allocated list. Lastly, the memory pool calls the general-purpose memory allocation function if the allocated memory pool is not large enough to service the user request.

Table 5: Performance of Xerces-C++ with a custom memory allocator and default memory allocator.

Xerces-C++ Method	Avg. Process Time
w. default memory allocator	159 ms
w. custom memory allocator	155 ms

We measured the overall processing time for the XML file using the default memory allocator and the custom memory allocator. As presented in Table 5, even when we plugged in the custom memory allocator we could not observe much performance gain (as small as 2.5%). This is an indication that Xerces-C++ does not exhibit excessive dynamic memory allocations. Figure 11 shows the frequency-lifetime diagram for our experiments. In this figure, none of the allocation objects reside in the high-frequent, short-lifetime region of the graph. EMAD therefore does not report any excessive dynamic memory allocations.

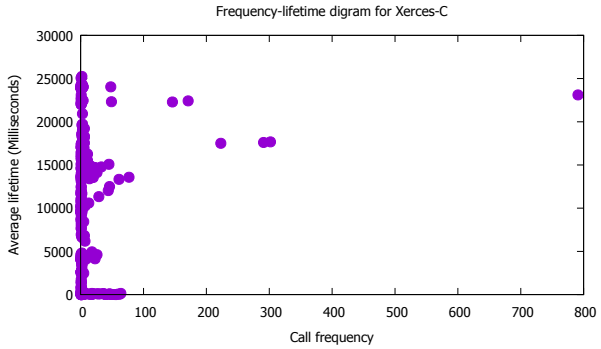

Figure 11: Frequency-lifetime ratio chart for Xerces-C

The frequency-lifetime ratio chart in Figure 12 also confirms our finding. The range of values for frequency-lifetime ratio is as low as 16. In other applications where we found excessive dynamic memory allocations, this ratio has a range as high as $150,000$.

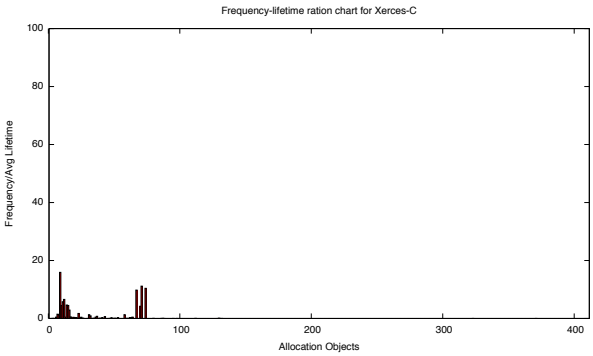

Figure 12: Frequency-lifetime ratio chart for Xerces-C

4.6 Discussion of results and threat to validity

Our experiments show the validity of EMAD's approach. EMAD was able to correctly detect and locate when a software application

has, or does not have, excessive dynamic memory allocations. This kind of analysis will help software developers resolve excessive dynamic memory allocations faster. More importantly, it will eliminate the laborious process of detecting the anti-pattern via manual source code analysis.

The main advantage of the clustering technique over the outlier detection method is it does not categorize allocation objects as excessive dynamic memory allocations when it has a low frequency. In the outlier detection technique, because we consider frequency-lifetime ratio as the analytical value, it can still report extreme outliers when the frequency is low and lifetime of the allocation object is short. These low-frequent and short-lifetime values may sometime beat some high-frequent, short-lifetime objects. With the clustering technique, this kind of false positive is not possible.

When using the clustering technique, EMAD's users have to provide the number of clusters to use in the analysis phase. There are some advanced data mining techniques [22, 28] for learning this parameter from the dataset itself. EMAD, however, does not employ those techniques at the moment. Unfortunately, this can cause EMAD to provide incorrect predictions if the user does not specify a reasonable number of clusters. If the dataset has very clear separable partitions, then the impact of this parameter can still be mitigated. On the other hand, when using outlier detection technique, users have to provide μ, which may need some trial and error.

When using the clustering technique, EMAD performs quantitative analysis and detects excessive dynamic memory allocations only if high-frequent and short-lived allocation objects resides in the same cluster. A software developer, however, may still think that there are excessive dynamic memory allocations in other clusters by looking at the numbers. In this situation, EMAD's prediction may not be inline with software developer's expectation. EMAD, however, can still be helpful because the software developer can manually analyze the frequency-lifetime diagram or the frequency-lifetime ratio chart to understand the big picture. A recommended way for further analysis is to do a comparative analysis of both two-dimensional and one-dimensional datasets.

5. RELATED WORK

Automated approaches for detecting excessive dynamic memory allocations cannot be found in literature. Likewise, existing approaches for detecting software performance anti-patterns have categorized excessive dynamic memory allocations as an undetectable software performance anti-pattern [6–8]. Although there are several approaches for detecting memory leaks and memory access errors using DBI [29], the excessive dynamic memory allocation problem has not been attacked by the research community.

Chen et al. [30] have developed a tool called *MemBrush* that can be used to detect memory allocation/deallocation functions using DBI in stripped binaries. Their approach is useful in detecting memory leaks and memory access errors, but they do not discuss detecting excessive dynamic memory allocations. However, by combining the *MemBrush* approach with our approach, it may be possible to relax our assumptions about allocation/deallocation routines as EMAD expects a particular signature for those routines.

Lu et al. [31] have developed a tool called PerfBlower, which can be used to detect memory related performance problems. They have developed a domain specific language called *Instrumentation Specification Language (ISL)* that is used to specify the memory related performance issues. The application code is executed on top of a modified Java Virtual Machine (JVM) where ISL is used to modify the JVM. Although they have tried to detect several memory related performance issues, excessive dynamic memory allocation anti-pattern, related to allocation object's life time has not been considered. Moreover their approach requires recompilation of the JVM whereas our approach does not require any kind of recompilation of the target system.

DBI has been used to identify other root causes of performance anomalies. For example, Attariyan et al. [32] proposes an approach to detect root causes of performance anomalies, such as misconfigurations, using DBI. Menon et al. [33] uses DBI to diagnosis performance overheads in Xen virtual machine environments. The root causes they try to detect are related to I/O handling in virtual machine environments—particular related to TCP connections.

There are research efforts on finding the object life times in managed languages (*e.g. Java*) [34]. In managed languages the garbage collection process can happen at anytime, therefore the timestamp at which an object is deleted cannot be used alone to approximately calculate the object lifetime. We believe that by integrating precise object lifetime calculation techniques, we can extend our technique for applications created using managed languages.

6. CONCLUDING REMARKS

This paper discussed our work on a tool called EMAD, which can detect excessive dynamic memory allocations software performance anti-pattern. Our experience and results show that EMAD can correctly report the locations where the software application is performing excessive dynamic memory allocations. Based on experience gained from applying EMAD to several widely used open-source software applications, we have learned that DBI can serve as a good platform for detecting software performance anti-patterns. We therefore plan on applying DBI to detect other software performance anti-patterns [3], such as God Class, Single Lane Bridge, and Circuitous Treasure Hunt. Likewise, EMAD's current technique works only with C/C++ software applications. We plan to investigate if EMAD's approach will work on applications written in interpreted languages like Java, Python, PHP, and JavaScript.

EMAD is available in open-source format and has been integrated into the Pin++ distribution: github.com/SEDS/PinPP.

7. REFERENCES

[1] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," in *Memory Management*. Springer, 1995, pp. 1–116.

[2] D. Detlefs, A. Dosser, and B. Zorn, "Memory allocation costs in large c and c++ programs," *Software: Practice and Experience*, vol. 24, no. 6, pp. 527–542, 1994.

[3] C. U. Smith and L. G. Williams, "Software performance antipatterns." in *Workshop on Software and Performance*, 2000, pp. 127–136.

[4] ——, "More new software performance antipatterns: Even more ways to shoot yourself in the foot," in *Computer Measurement Group Conference*, 2003, pp. 717–725.

[5] J. Din, A. B. Al-Badareen, and Y. Y. Jusoh, "Antipatterns detection approaches in object-oriented design: A literature review," in *Computing and Convergence Technology (ICCCT), 2012 7th International Conference on*. IEEE, 2012, pp. 926–931.

[6] C. Trubiani and A. Koziolek, "Detection and solution of software performance antipatterns in palladio architectural models." in *ICPE*, 2011, pp. 19–30.

[7] V. Cortellessa, A. Di Marco, and C. Trubiani, "Performance antipatterns as logical predicates," in *Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on*. IEEE, 2010, pp. 146–156.

[8] J. Xu, "Rule-based automatic software performance diagnosis and improvement," *Performance Evaluation*, vol. 67, no. 8, pp. 585–611, 2010.

[9] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[10] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Applied statistics*, pp. 100–108, 1979.

[11] E. D. Berger, B. G. Zorn, and K. S. McKinley, "Oopsla 2002: Reconsidering custom memory allocation," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 46–57, 2013.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[13] N. Moha, "Detection and correction of design defects in object-oriented designs," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM, 2007, pp. 949–950.

[14] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan Notices*, vol. 42, no. 6, pp. 89–100, 2007.

[15] J. H. Hill and D. C. Feiock, "Pin++: an object-oriented framework for writing pintools," in *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*. ACM, 2014, pp. 133–141.

[16] B. G. Ryder, "Constructing the call graph of a program," *Software Engineering, IEEE Transactions on*, no. 3, pp. 216–226, 1979.

[17] F. Eichinger, K. Böhm, and M. Huber, "Mining edge-weighted call graphs to localise software bugs," in *Machine Learning and Knowledge Discovery in Databases*. Springer, 2008, pp. 333–348.

[18] I. Corporation, "Pin 2.14 User Guide," https://software.intel.com/sites/landingpage/pintool/docs/67254/Pin/html/.

[19] E. D. Berger, B. G. Zorn, and K. S. McKinley, "Composing high-performance memory allocators," in *ACM SIGPLAN Notices*, vol. 36, no. 5. ACM, 2001, pp. 114–124.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1997.

[21] C. Fraley and A. E. Raftery, "How many clusters? which clustering method? answers via model-based cluster analysis," *The computer journal*, vol. 41, no. 8, pp. 578–588, 1998.

[22] C. A. Sugar and G. M. James, "Finding the number of clusters in a dataset," *Journal of the American Statistical Association*, vol. 98, no. 463, 2003.

[23] V. J. Hodge and J. Austin, "A survey of outlier detection methodologies," *Artificial Intelligence Review*, vol. 22, no. 2, pp. 85–126, 2004.

[24] C. Leys, C. Ley, O. Klein, P. Bernard, and L. Licata, "Detecting outliers: do not use standard deviation around the mean, use absolute deviation around the median," *Journal of Experimental Social Psychology*, vol. 49, no. 4, pp. 764–766, 2013.

[25] M. Imran and H. Hlavacs, "Provenance in the cloud: Why and how," in *The Third International Conference on Cloud Computing, GRIDs, and Virtualization*, 2012, pp. 106–112.

[26] M. Cooperation, "Northwind database," https://northwinddatabase.codeplex.com/.

[27] SQLite, "Release History," http://www.sqlite.org/changes.html.

[28] S. Salvador and P. Chan, "Determining the number of clusters/segments in hierarchical clustering/segmentation algorithms," in *Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on*. IEEE, 2004, pp. 576–584.

[29] G. R. Luecke, J. Coyle, J. Hoekstra, M. Kraeva, Y. Li, O. Taborskaia, and Y. Wang, "A survey of systems for detecting serial run-time errors," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 15, pp. 1885–1907, 2006.

[30] X. Chen, A. Slowinska, and H. Bos, "Who allocated my memory? detecting custom memory allocators in c binaries," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*. IEEE, 2013, pp. 22–31.

[31] L. Fang, L. Dou, and G. Xu, "Perfblower: Quickly detecting memory-related performance problems via amplification."

[32] M. Attariyan, M. Chow, and J. Flinn, "X-ray: automating root-cause diagnosis of performance anomalies in production software," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*. USENIX Association, 2012, pp. 307–320.

[33] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel, "Diagnosing performance overheads in the xen virtual machine environment," in *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*. ACM, 2005, pp. 13–23.

[34] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović, "Generating object lifetime traces with merlin," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 28, no. 3, pp. 476–516, 2006.