GrowthTracker: Diagnosing Unbounded Heap Growth in C++ Software

Erik Hill*, Daniel J. Tracy†, and Sheldon Brown‡
Center for Hybrid Multi-core Productivity Research
University of California, San Diego

*Email: erikghill@gmail.com
†Email: daniel.joseph.tracy@gmail.com
‡Email: sgbrown@ucsd.edu

***Abstract*—Unbounded growth of heap memory degrades performance and eventually causes program failure. Traditional memory leaks are the most commonly recognized, but not the only cause of this issue. Large software systems use many aggregate data structures that can grow arbitrarily, and application behavior that produces unbounded growth of these structures is common. This growth can remain undetected by both memory leak and staleness detection tools. In this paper, we present an approach for reliably identifying aggregate data structures that can grow without bound over the lifetime of a program. Our solution tracks all aggregates over the lifetime of the program and utilizes heuristics to identify non-convergent growth. Our diagnostic method continuously reduces false positives and false negatives during execution, producing more accurate reports for as long as it is allowed to continue execution. In addition, we present techniques to utilize this method in large, pre-existing C++ software without requiring extensive code modification. Our tool identified data structures with this issue in Google's Chrome web browser and Apple's Safari browser among others.**

***Keywords*-C++; memory leak; memory tumor; memory bloat; data structures**

I. Introduction

A great number of software programs contain memory management errors that cause them to continually increase memory allocation over time, causing performance decline and eventual program failure. These problems can be difficult to detect during testing due to how long it can take to exhibit overt symptoms. They can also be difficult to diagnose, as the final allocation that crashes the program is not necessarily related to the error.

A well-known cause of this problem, known as a *memory leak*, is when the program fails to deallocate memory that is no longer referenced. This prevents the program from recycling unused memory for new allocations, resulting in increased allocations over time. Memory leaks are not relevant in garbage-collected languages, and are easily diagnosed by allocation-tracking tools in languages without dynamic garbage collection [7], [13], [16].

The term *memory leak* is typically expanded to include many other forms of memory mismanagement [3], [8], [12], [21], [22], with some authors using different additional adjectives to distinguish their properties [11], [14]. We find it useful to distinguish the common forms of memory mismanagement, and with greater brevity. When all references to a memory block are lost, we refer to this as a *memory leak*, as is traditional. However, it is sometimes the case that references are retained to a finite block of memory that is no longer needed for the remainder of the program. We refer to this as a *memory cyst*. If the program's behavior is such that retained memory blocks can accumulate without bound, we refer to this as a *memory tumor*. Observing that cysts generally will not cause programs to run out of memory, we focus solely on detecting and diagnosing tumors, which are of particular concern due to their threat to program stability.

Tumors by their nature must be an instance of an aggregate data structure which can reference a variable number of allocations or be reassigned to a variable size allocation. Examples of aggregates include sequences, trees, and hash tables. When an aggregate grows without bound during the course of a program's execution, it causes long-term instability similar to a memory leak. The fundamental cause could be anything from improper maintenance of the aggregate to a program design error. The methods presented here identify the specific aggregate instances that exhibit unbounded growth at run time, allowing software engineers to correct the problem in the context of their work.

We perform dynamic tests in which the target program is run continuously in an automated mode, cyclicly utilizing sample inputs and program features. During this process, the software performs periodic queries of each aggregate's size. In C++, this requires modifying the source code to track all aggregate data structures in

a program and to allow querying their size in a universal way. A heuristic is then used to identify tumors using the size history of aggregate instances. Our method reports periodically and becomes more accurate the longer it is allowed to run, reducing false positives and false negatives continuously.

II. Method Overview

While the target program is running, the tracking framework must maintain a data structure, the central aggregate tracker (CAT), containing references to all other aggregates in the system. When an aggregate is instanced, it must be added to the CAT, and it must be removed when it is deallocated. In addition, there must be a means for all aggregates in the system to respond to a query of its size during traversal of the CAT. This can take the form of dynamic binding, a common base class for all aggregates, or a common abstract interface.

While the application is being tested, the CAT is periodically traversed in order to generate a snapshot of the size of the program's aggregates, which serves as input data to the heuristic. We provide a detailed implementation that requires minimal source code modification for C++ programs in Section III.

In addition to maintaining the CAT structure, the target program must be setup to run in a continuous, automated test mode. This mode executes the program indefinitely in order to exercise the code paths and data structures in the system. Its design is application-specific, but we provide suggestions in Section V.

The final component of the framework is the heuristic used to detect unbounded growth, discussed in Section IV-B.

Although the method described is applicable to other languages, our framework, GrowthTracker, was designed and implemented in the context of C++, and many of the problems and solutions herein will be specific to that language. The framework was developed in order to diagnose a memory management problem in a complex C++ software installation intended for several months of continuous execution without intervention. In addition, we have used this framework to diagnose unbounded heap growth problems in several widely-used software systems such as the Chrome [17] and Safari [18] browsers, the Ogre3D graphics library [15], and others. Our testing indicates that this problem is very widespread, as almost all of the software tested exhibited some problems with unbounded heap growth.

III. C++ Implementation

Our design is dictated by the constraint that manual modification of C++ source code is infeasible for large systems. Driven by this constraint, this section discusses injecting size tracking behavior into all aggregates in the target software.

A. Aggregate Behavior Injection

In order to inject additional behavior into aggregates, we employ two strategies: the first applies to preexisting libraries such as STL and boost [4] containers, while the second is more convenient for custom aggregates that developers can directly modify.

Dealing with library aggregates that should not be modified requires developers to switch aggregate types being employed in their software. We have predefined small wrapper classes for STL and boost containers that derive from those containers and provide the additional tracking functionality. Utilizing them merely requires a textual search-and-replace to alter include and namespace declarations referring to these containers. The majority of the wrapper class' implementation is universal for all aggregates, and uses multiple inheritance to provide additional functionality transparently. If desired, developers can reuse this pattern for additional custom aggregates by writing more wrapper classes following a similar pattern. Alternatively, custom data structures can simply be derived from our base tracking class, which will provide similar functionality without type or namespace declaration changes.

Note that, using either method, the development work required to integrate these tools is a small modification to each aggregate type being utilized. Due to software reuse, large projects tend to have few unique custom data structures, making this task more or less independent of the size of the project itself. In addition, once these changes are made, GrowthTracker is designed to be turned on as needed as a compile time option without incurring this development time cost again (Listing 4).

B. Aggregate Tracking

In this system, every instance of an aggregate will add itself to the CAT upon construction and remove itself upon destruction. This is done automatically by the $tracked\_base$ base class (Listing 1), which will also hold information on instantiation order and the specific type of the aggregate. This is the identifiable information we are able to gather for each aggregate quickly and portably. This base class also provides a function object for querying the number of elements held in the aggregate.

The $tracked\_impl$ class binds the aggregate type to the *tracked_base* class through multiple inheritance, and provides some additional functionality (Listing 2). The wrapper class for each aggregate derives from

$tracked\_impl$ in order to reuse its functionality and keep the wrapper as simple as possible. Each wrapper must implement constructors that simply initialize the $tracked\_impl$ base class with the same parameters. The $tracked\_impl$ base class uses templated constructors for generality. Aggregate wrappers can potentially use templates to ease the task of overriding all constructors, but must explicitly override all one-argument constructors to avoid unexpected type conversions and infinite recursion, in the copy constructor case. New aggregates must then use the new *tracked* namespace in their type definitions. This three layer approach is demonstrated in Listings 1 - 4, which present an example implementation of a tracked STL map class. The code shows a simplified implementation that excludes support for multithreading, and call stack tracing options. Additional implementation options are discussed below.

1) C++11 Feature Alternatives: C++11 provides the function and bind implementations presented in $tracked\_base$ and $tracked\_impl$, which are used to provide call-back functionality for querying aggregate size from the CAT. This section presents a couple of alternatives to this reliance on C++11. The boost libraries [4] provide the necessary functionality with their function0 and bind implementations. These implementations are header based and require no linking, but nevertheless create a dependency on third party code.

Another solution is to use a pure virtual *size* function in the $tracked\_base$ class which is implemented in the $tracked\_impl$ class by querying the overridden data structure. In practice, this is the approach we used most often, since the majority of the open source projects we targeted were not configured for C++11. However, this solution can cause compilation problems with the non-standard practice of using incomplete types as template arguments for some standard containers [1]. The forward declaration of template arguments for standard containers, such as vector, is not officially supported by the C++ standard. However, most compilers support the practice, and the desire to reduce compile times has encouraged its inclusion into some software projects. Once the *tracked* wrapper instance replaces the original data structure, the forward declaration of the template argument is no longer sufficient for compilation, and the argument's definition must be included. In practice, this side-effect can defeat our goal of requiring minimal non-automated source code changes.

2) Aggregate-Identification Options: Using portable C++ language features, combined with the restriction against non-automated source code modifications, we are limited to identifying aggregates by their full type and instantiation order. We have found this to usually be sufficient, but applications that reuse specific aggregate types extensively and exhibit non-deterministic instantiation behavior may require more specific data. Non-portable techniques to acquire this information without non-automated source code modification exist, although these impose some run-time overhead.

We have used $StackWalk64$ on Windows and $backtrace$ with gcc to gather run-time call stack information. The performance implications of gathering and storing this additional metadata depend on the frequency of aggregate creation in a particular piece of software. Section VI-D compares the run-time overhead of various stack tracing options for some high-performance software.

IV. Growth Tracking & Detection

In this section we build upon the aggregate tracking foundation in order to detect unbounded growth among all the aggregates present in the target software. We first offer a naive approach to growth detection that demonstrates the power of the CAT. We then present a growth tracking heuristic that vastly improves on the naive approach in several ways.

A. Naive Growth Detection

With access to the CAT, a simple approach to identify growing aggregates follows.

```
Every 5 minutes do:
  for each agg in CAT:
    if agg.query_size() > agg.previousSize:
      Print agg's metadata
    agg.previousSize = agg.query_size();
```

This method, when coupled with a cyclic test, identified tumors in the software we targeted. However, this naive heuristic has the following problems: 1. Comparing with previous size will detect growth caused by size oscillation. 2. A constant time period will only catch a slow-growing tumor periodically. 3. Finitely-growing aggregates will be reported many times.

The prevalence of false positives and negatives can be greatly reduced by the heuristic given in Section B.

B. Reporting Heuristics

The goal of the testing heuristic is to identify data structures that grow without bound during a program's execution. Given a series of size samples for all aggregates in the program, there are many potential heuristics to employ using rate of growth, consistency of growth, size, number of growth events, etc.

We propose analyzing a data structure's history as consisting of two intervals. In the initial interval, the

Listing 1: Aggregate base class maintains CAT

```
#include <functional>
static int alloc_count = 0;
static std::set<tracked_base*> CAT; // simple CAT implementation
struct tracked_base{
    std::string typeName;
    int id;
    int sizeMaximum;
    std::function<int (void)> sizeFunc;
    tracked_base( const char * _typeName, std::function<int (void)> sizeFunc_ )
        : typeName(_typeName), id(alloc_count++), sizeMaximum(0), sizeFunc(sizeFunc_){
        CAT.insert( this );
    }
    ~tracked_base(){ CAT.erase( this ); }
    int query_size() const{ return sizeFunc(); }
    void operator=( const tracked_base& ){}
};
```

Listing 2: *tracked_impl* **derives from type of aggregate and** *tracked_base*, **and provides query functionality**

```
// this example assumes Type has a size() function.
// a full implementation includes multiple argument constructors.
#define TYPE_STRING(Type) typeid(Type).name() // avoid RTTI with __PRETTY_FUNCTION__
template<typename Type>
struct tracked_impl : public Type, public tracked_base{
    tracked_impl() : tracked_base(TYPE_STRING(Type), size_func()) {}
    tracked_impl( const tracked_impl<Type>& rhs )
        : Type(rhs), tracked_base(TYPE_STRING(Type), size_func()) {}
    template<typename Arg1>
    tracked_impl( Arg1 a1 )
        : Type( a1 ), tracked_base(TYPE_STRING(Type), size_func()) {}
    int type_size() const{ return (int) ((Type*)this)->size(); }
    std::function<int (void)> size_func(){
        return std::bind( &tracked_impl<Type>::type_size, this );
    }
};
```

Listing 3: Need wrapper for each specific aggregate type: tracked::std::map partial definition shown

```
// example tracked stl structure (map).
namespace tracked{
    namespace std{
        template<typename K, typename V, typename P = ::std::less<K> >
        struct map : public tracked_impl< ::std::map<K, V, P> >{
            map() {}
            map( const map<K, V, P>& rhs ) : tracked_impl< ::std::map<K, V, P> >(
                  (const tracked_impl< ::std::map<K, V, P> >&)rhs ) {}
        };
    }
}
```

Listing 4: Namespace switch to enable tracking

```
#if TRACK_ALL
#define trackable tracked       // trackable::std::map --> tracked::std::map
#else
#define trackable               // trackable::std::map --> ::std::map
#endif
```

aggregate is expected to change size, potentially increasing and decreasing many times. However, for non-malignant aggregates, the maximum of its size samples over time should stabilize to a finite value. In the second interval, the aggregate proves its stability by not growing higher than its previous maximum size at the end of the initial interval. Growth in the second interval above the previous maximum size indicates a potential tumor. The use of the highest sampled size, as opposed to methods that use last sampled size [12], is imperative to eliminate false positives over time.

With this in mind, we use a simple, two-sample comparison approach: each sample correlates to the end of one of the two intervals. If an aggregate's size in the second sample is higher than the maximum established for that aggregate during the first sample, an aggregate is a tumor candidate. False positives are data structures that are reported as having unbounded growth when, in fact, their growth will be bounded. False negatives are data structures that have unbounded growth but are not reported. A larger initial interval reduces reported false positives, as it allows more aggregates to achieve maturity and a more stable maximum size value. A larger second interval reduces false negatives by detecting rarer growth events and slower rates of unbounded growth. Smaller interval sizes have the advantage of providing immediate feedback and convenience for the developer.

Rather than forcing the developer to choose interval sizes, making immediate decisions on running time versus accuracy level, we propose using a continuous test in which accuracy improves over time, yet immediate feedback is also available. In our solution, samples are taken with an exponentially growing interval size, with reports being generated at a minimum time (with lower accuracy) and at each sample thereafter at ever higher accuracy. Each time a new sample is taken, a report is generated at the current level of accuracy. Then, the old second interval becomes the initial interval in a new, more accurate test. Using this method, the initial interval size can be small and the choice of initial interval size is insignificant to the effectiveness of the method.

We report on any aggregates that have grown beyond their previous maximum size in the second interval. Young aggregates are excluded by only reporting data structures with three samples present, ensuring that they exist for the full two intervals required for an accurate test. As the test continues, results will exclude more false positives (due to the increasing initial interval size) and false negatives (due to the increasing second interval size). In fact, tumor aggregates can escape detection only so long as they increase size less often than the time represented by the second interval. Please refer to Listing 5 for pseudo code of this heuristic.

In addition to reporting data structures that have grown in the second interval, we also report more detailed information on the history of the aggregates to help focus attention on the most important tumors. This information includes whether the aggregate had also grown beyond its maximum size in the previous interval, the number of intervals in which the aggregate grew beyond its maximum size (to confirm continuous growth), and the previous and current size of the structure (yielding its rate of growth). To assist in diagnostics, we also output the full type of the aggregate and its instantiation order.

1) Tumor Identification Accuracy: Our experience indicates that our heuristic accurately identifies all tumors within minutes for most programs. Longer running times may be necessary in more rare cases. An exceptional case, the Chromium project [17], is detailed in Section VI.

We give developers some control over the prevalence of false positives by making the test more accurate over time, and by the detailed history output for each candidate. Developers can determine how quickly to intercede based upon the confidence level they want to achieve before investigating a tumor candidate. In our experience, we have found that an aggregate with two consecutive reports using an exponentially increasing sample duration has been sufficient to identify true tumors within minutes. However, in some cases false positives can remain for some time, depending upon the role of the aggregate. The most common cause of this is a resource pool, cache, or buffer with a large, but finite size policy. In such cases, the aggregate will have to fill before it is no longer suspect.

False negatives may theoretically also remain for some time in the case of tumors that grow very slowly. We have not encountered such a case in practice. For long running tests, the impact of a false negative that grows very slowly will be low. The faster-growing tumors most likely to cause program failures are detected more quickly.

It should also be noted that the tracking system cannot detect tumors from poor tests that do not exercise all application behaviors. Therefore, tumors may still exist in software that isn't tested properly. This is true of all dynamic leak and tumor detection tools.

V. Automated Application Testing

An automated cyclic test is essential for GrowthTracker to maximize reporting accuracy. The test should

Listing 5: Pseudo Code for Growth Tracking Heuristic

```
origTimeBetweenSamples = 120; //seconds (adjust as desired)
function GrowthTrackingUpdate() {
    enforcedDurationBetweenSamples = origTimeBetweenSamples*(2^numSamples);
    durationSinceLastSample += GetTimeDurationSinceLastUpdate();
    if( durationSinceLastSample >= enforcedDurationBetweenSamples ) {
        GrowthTrackingSample();
        durationSinceLastSample = 0;
        ++numSamples;
    }
}
function GrowthTrackingSample() {
    for each aggregate in CAT { // CAT defined in Listing 1
        aggregate->sampleCount++;
        if( aggregate->query_size() > aggregate->sizeMaximum ) {
            if( aggregate->sizeMaximum > 0 and aggregate->sampleCount > 2)
                Report aggregate;
            aggregate->sizeMaximum = aggregate->query_size();
        }
    }
}
```

represent complete coverage over the application's functionality. If all code paths are not included in the complete automated test, there may be problems that remain undetected by the test. Clearly, a memory problem cannot be located if it is not exhibited in the test. A test that exercises all functionality of an application equally is much more likely to produce good diagnostics quickly than an unbalanced one. There may, of necessity, be events that occur less frequently, but ensuring that they occur cyclically is necessary for GrowthTracker to find allocation problems with those systems eventually.

Another factor that is just as important: the developer integrating the test must not miss any aggregate data types. Every common type that can grow in allocation size (such as std::string, any STL or boost aggregate, etc.) should be included in the automated namespace replacement, and any custom data structures that can grow likewise requires a similar wrapper or must be modified to derive from $tracked\_base$. Any aggregate types that are missed will not be detected if they exhibit unbounded growth.

VI. Results

GrowthTracker was developed to solve memory mismanagement issues in the Scalable City project, a 3D visual artwork demonstration in the form of a multi-user virtual world [5]. We created automated avatars to traverse the virtual world and randomly chose from all available actions to perform at each step along the way. Our memory growth issues were caused by a rare combination of events. Nevertheless, after over 24 hours of automated testing, GrowthTracker reported the specific data structures exhibiting unbounded growth. Only the type and instantiation order of aggregates were reported, but this was sufficient to identify the offending data structures in minutes.

For further testing of our methods, we applied GrowthTracker to several public software projects: the Ogre3D graphics engine, the Bullet physics library, WebKit, and Google's Chromium browser.

A. Ogre3D & Bullet

GrowthTracker identified a memory tumor in the instancing module of Ogre3D [15], a popular open source graphics rendering engine. This bug, which caused tumors in GPU memory allocation, had been present since early 2007 and is therefore present in many games and other programs. Ogre3D maintainers have integrated our fix for this bug into the software.

Our tests of the Bullet physics engine [6] dealt with a subset of Bullet's available features, but revealed no growth anomalies in the core physics engine after long automated tests. GrowthTracker did, however, locate a tumor in the OpenGLSupport library which provides visualization for the Bullet Demos. Overall, the result of the tests serve to increase confidence in the stability of software developed with the library, which itself is a valuable result.

B. WebKit

WebKit [19] is a widely used web browser engine (Safari, iPhone, iPad, Android, Kindle). It utilizes custom data structures exclusively. We created wrappers to track 14 data structure types including Vector,

HashTable and DoublyLinkedList. Then, we modified WebKit's MiniBrowser project to cycle through 100 popular websites every 20 seconds with an initial sample time of 5 minutes. Over the lifetime of the test, only three aggregates were reported. Two aggregates were reported immediately: a Vector and a HashTable of WebBackForwardListItem. The Vector reached a cap of 100 elements by the third sample and was never reported again. The HashTable continued to grow at a rate of one element per web page navigation; a definite tumor. A search over the source code for the HashTable's signature immediately identified the offending aggregate in the WebProcessProxy class of the UIProcess module. Upon inspection, we realized that some infrastructure existed to propagate remove calls from the Vector to the HashTable in an attempt to enforce the same 100 element limit, but these elements were never removed. At the time of this writing, we have verified that the Safari browser uses the WebProcessProxy class of WebKit and therefore also contains the tumor.

The third aggregate appeared intermittently in the first few samples, and then consistently in later samples, but stayed very small, reaching only 38 elements after 21 hours. This aggregate was a tumor whose slow growth was due to our insufficient testing strategy. Some of the websites in our test occasionally launched a pop-up window which was blocked, but the identified aggregate stored a window handle to each one and never removed them. This demonstrates the virtue of tracking small growing potential tumors which others ignore [9], [12]. Additionally, it shows the usefulness of our exponentially growing interval size, which consistently reported this tumor after the first few samples even though it was growing infrequently. If a user frequently navigates to web pages with pop-up windows, then the tumor will grow quickly and can cause serious problems. This tumor appears to only affect the MiniBrowser and not the WebKit core. Tumors may remain in WebKit from the use of custom aggregates that we may have missed in the wrapper creation stage. It should be a trivial task for the WebKit developers to identify the custom aggregate types utilized in their system.

C. Chromium

The Chromium project is the open source foundation of Google's Chrome browser. GrowthTracker has detected 27 tumors in Chromium. We performed a nearly complete conversion of aggregates in Chromium svn revision 109290 (November 2011). Our automated test involved cycling through 1000 popular websites, loading a new site every 20 seconds. Our initial sample size was five minutes. Our initial results are shown in Figure 1.

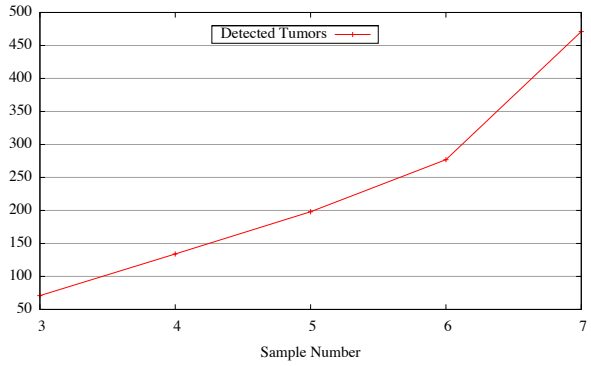

Figure 1: GrowthTracker analyzing Chrome browser

While GrowthTracker normally refines a set of candidate tumors that quickly becomes stable, this test yielded an ever-increasing set of candidates. The problem exhibited is due to specific behaviors of aggregates within aggregates in this software.

Take the example of a linear hash table [10] implemented using dynamic arrays (e.g. std::vector) for each bucket. Since our method tracks aggregates individually, each vector is tracked in addition to the parent sequence of buckets. In such a structure, the hash table is optimized to keep the average number of elements in each bucket small by growing the sequence of buckets as necessary and rehashing some elements. This causes random and rare growth and shrinking perturbations in each vector, as well as allocation of new vectors as the hash table grows. This is similar to the case being dealt with in Figure 1. It should be noted, however, that the metadata output makes it obvious that these aggregates are small and have grown very little, though the graph does not reflect this as it only indicates which aggregates grew in the second interval.

Another potential problem is simply an ever-increasing number of dynamically generated tumors. While technically tumors, dealing solely with parent aggregates is more productive.

We attack these problems in two ways. First, we apply an additional filter in which the expected growth rate must be proportional to the interval size. This eliminates small growth events in sub-aggregates caused by adding elements to its parent aggregate. Secondly, we consolidate aggregates with the same type string, which collapses potentially many tumor sub-aggregates to a single virtual instance, but may also hide separate tumors with a duplicate type in some cases. The results are shown in Figure 2. While useful in the present

framework, more direct solutions are discussed in Section VIII.

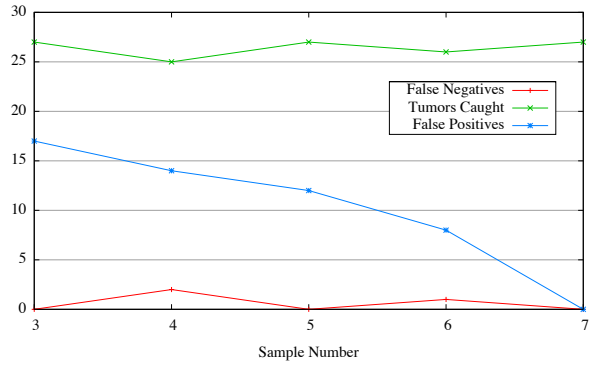

Figure 2: Chrome after corrective post-processing

Output from the last sample (sample #7) is verified correct manually. In each sample, false positives indicate additional entries not present in the last sample, while false negatives indicate entries missing compared to the last sample. It took ten and a half hours of automated testing to diagnose the 27 tumors present in Chromium.

While most applications of GrowthTracker are relatively simple, Chromium illustrates potential complications, and methods that may be useful in some cases. These are recommended only if they are clearly needed after initial testing. Nevertheless, GrowthTracker successfully identifies a large number of tumors in Chromium which have been verified by source code inspection. A fix for the fastest-growing tumor has been accepted by the maintainers of the Chromium code base, further confirming our results. This tumor was present since the initial release of Chrome in 2008.

D. Overhead

The run-time overhead of our tested aggregate-identification options for two high-performance subsystems are given in Table I. These times include both aggregate construction and destruction overhead and scanning operations, though scanning operation overhead is insignificant. Tests were performed on an Intel i7 950 machine with 6 GB memory running 64-bit Windows 7 at 3.07 GHz.

The Ogre3D test was performing skeletal animation, while the Bullet test was performing physical simulation on a large number of rigid bodies. Note that the overhead of our methods sans stack tracing, which we have found viable for most software in practice, is insignificant. The use of stack tracing, however, can impose significant overhead for high-performance, real-time software. The additional time is primarily caused by the stack tracing steps during aggregate construction, while the space overhead is mainly due to debugging symbols loaded during execution.

VII. Related Work

The majority of related previous work in this area has focused on garbage collected languages such as Java. Several approaches take advantage of the garbage collection framework or the JVM to detect memory cysts and tumors [3], [9], [12], [20]. The great advantage of utilizing the JVM is that it provides information about the current Objects in memory and their interrelationships without the need for source code modification. Native C++ has no such luxury.

LeakBot [12] is the most closely related previous work. LeakBot requires two temporally spaced snapshots that consist of a list of every live Object (Integer, Character, etc.) in the system including the outgoing references for each object. It builds two directed object reference graphs from these lists and compares the graphs to identify data structures that are potentially growing. The authors employ some elegant metrics to determine the root Object of one or more tumors by considering the age and relationship of child elements. In their live mode, a tracing agent monitors the child elements of the identified potential tumors via the JVM for changes in behavior.

Auto-detection of root aggregates (including custom aggregates and standard Collections), is a virtue of LeakBot that would allow GrowthTracker to eliminate the reliance on the developer to identify custom aggregates and eliminate child aggregate output. However, LeakBot's weaknesses include: ignoring all new growth after the second snapshot, some potential tumors are eliminated by their metrics, and reports include false positives and regions that are not growing. Generating the live Object list that LeakBot relies on in native C++ applications would require deriving all objects from a common base class, building a reference graph between all objects and boxing native types into objects. It is simply infeasible to apply LeakBot to native C++ applications.

There are a variety of available tools to identify memory leaks in non-garbage-collected languages such as C++ [7], [13], [16]. Both memory cysts and tumors are more difficult to detect than leaks, and few authors address the problem in the context of static languages such as C++ [8], [14]. These papers focus on the more difficult general case of both cysts and tumors utilizing staleness detection. They do not require source

	Execution Time Increase		Virtual Memory Usage Increase	
Conditions	Bullet Physics	Ogre Graphics	Bullet Physics	Ogre Graphics
No Stack tracing	$< 1\%$	$< 1\%$	2.7%	4.87%
Stack tracing (4)	96.3%	41.1%	28.7%	64.0%
Stack tracing (6)	126.8%	102.2%	28.8%	64.8%

Table I: Time and space overhead for call stack tracing at call depths 4 and 6

code modification, but only succeed in detection in specific circumstances, and suffer from high rates of false positives or false negatives.

Staleness detection [2], [8], [14], [21] is the state-of-the-art approach for detecting cysts and tumors in C++ programs. It detects allocated memory that is not accessed for long periods of time. This approach has been successful in tracking down some cysts and tumors, but the scope of possible problems it can find is limited by design. It is often the case that programs periodically access the very memory that is causing unbounded growth. For example, an aggregate that exhibits unbounded growth could often be traversed for maintenance, causing accesses to all elements. Another problem with these methods for C++ software is that they diagnose staleness at the level of memory allocations. In C++, several commonly used aggregates can hold a large number of objects by value within a single allocation. A single allocation can, therefore, contain both 'live' and 'stale' objects. Dynamic arrays, which are commonly-used, high-performance sequences, exhibit both problems, as all elements are contained within a single allocation and their copy-on-resize behavior causes all elements to be accessed. Unbounded aggregate growth can occur in many ways, and does not necessarily imply discrete memory allocations that remain unaccessed. GrowthTracker detects unbounded growth regardless of memory access patterns.

VIII. Future Work

The method presented for detecting memory tumors is effective, and together with traditional memory leak detection tools covers *almost* all cases of unbounded memory loss in C++ software. However, our method does have several weaknesses that could be addressed more thoroughly.

A weakness of the presented method is its treatment of all aggregates equally, especially in the case of aggregates containing other aggregates. This causes anomalous reporting behavior for some structures as shown in the case of the Chromium project. It also causes a rare case in which a tumor may not be detected at all: if a parent aggregate grows by removing its child and replacing it with a new aggregate of a larger size, that growth will not be detected by our method.

A more accurate solution would be hierarchical in nature, assigning the size of child aggregates to their parent cumulatively. This may be difficult to implement, would impose additional conversion requirements upon target aggregates before testing, and may involve additional trade-offs such as significantly increased sample time. Nevertheless many of these faults could be minimized or used only in certain circumstances, and would result in more accurate and universally useful output.

Another weakness is the dichotomy of choice between lack of complete resolution on identified tumors and the high run-time overhead of stack-tracing operations. While our fast method provides the complete type of the aggregate and instantiation order, it may not be enough information to locate some aggregates. The stack-tracing techniques incur an additional run-time cost but provide full source code location. The two methods can be combined into a system that provides complete resolution with fairly low overhead by automatically excluding most stack-allocated aggregates from stack-tracing operations, utilizing aggregate lifetime prediction using available history, and providing a type-based selection functionality to accurately locate known tumors in a secondary test.

We are designing a multi-layer CAT implementation that will improve the potential performance impact of tracking in a multithreaded environment, as access to the CAT must be locked for aggregate construction and destruction. This issue is only significant if aggregates are being instantiated often across multiple threads, causing threads to wait upon one another to maintain the CAT.

In order to aid diagnostics, stack traces of insertion operations could be performed on identified tumors dynamically after their detection. This would require overriding insertion methods for custom aggregates, but could be done automatically for common library aggregates (e.g. STL, boost).

Finally, a C++ parser could be used to automate the identification of custom aggregates. While we expect developers to have a good grasp of custom aggregates

in their software, further automation would reduce the potential of incomplete coverage of aggregates.

IX. Conclusion

While traditional allocation-tracking tools are effective in diagnosing memory leaks, continual memory loss can still occur without leaks due to aggregate growth. We've introduced methodologies and a framework, GrowthTracker, to detect and diagnose unbounded aggregate growth in C++ software using automated tests. The methods are very effective, generally having a continually decreasing false positive rate and a continually decreasing false negative rate as the tool continues running. Our method directly tests unbounded growth, and does not depend upon staleness detection or other techniques that miss many cases. Our framework operates within the confines of the portable C++ language, and requires some source code modification, though this modification is mostly automated.

Acknowledgment

This work was supported by the UCSD Center for Hybrid Multicore Productivity Research, a National Science Foundation sponsored industry-university cooperative research center; IBM; Intel; the California Institute of Information Technologies and Telecommunications (Calit2); and the Center for Research in Computing and the Arts at UCSD.

References

[1] M. H. Austern. Containers of incomplete types. *C/C++ Users Journal*, February 2002.

[2] M. D. Bond and K. S. McKinley. Bell: bit-encoding online memory leak detection. In *Proc. 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-XII, pages 61–72. ACM, 2006.

[3] M. D. Bond and K. S. McKinley. Leak pruning. In *Proc. 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '09, pages 277–288. ACM, 2009.

[4] Boost C++ libraries. boost.org.

[5] S. Brown, K. Kho, K. Lee, and E. Hill. Accelerating the scalable city. *Concurrency and Computation: Practice and Experience*, pages 2187–2198, 2009.

[6] Bullet physics library. bulletphysics.org.

[7] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proc. Winter 1992 USENIX Conference*, pages 125–138, 1991.

[8] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Proc. 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-XI, pages 156–164. ACM, 2004.

[9] M. Jump and K. S. McKinley. Cork: dynamic memory leak detection for garbage-collected languages. In *Proc. 34th Symposium on Principles of Programming Languages*, POPL '07, pages 31–38. ACM, 2007.

[10] W. Litwin. Linear hashing: a new tool for file and table addressing. In *Readings in database systems*, pages 570–581, San Francisco, CA, USA, 1988. Morgan Kaufmann Publishers Inc.

[11] J. Maebe, M. Ronsse, and K. D. Bosschere. Precise detection of memory leaks. In *International Workshop on Dynamic Analysis*, WODA '04, pages 25–31, 2004.

[12] N. Mitchell and G. Sevitsky. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large java applications. In *Proc. 17th European Conference on Object-Oriented Programming*, ECOOP '03, pages 151–172. Springer Berlin / Heidelberg, 2003.

[13] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proc. 2007 Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100. ACM, 2007.

[14] G. Novark, E. D. Berger, and B. G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *Proc. 2009 Conference on Programming Language Design and Implementation*, PLDI '09, pages 397–407. ACM, 2009.

[15] Ogre: Object-oriented graphics rendering engine. ogre3d.org.

[16] Parasoft Insure++. parasoft.com/insure.

[17] C. Reis, A. Barth, and C. Pizano. Browser security: lessons from google chrome. *Communications of the ACM*, 52:45–49, Aug. 2009.

[18] Safari web browser. apple.com/safari.

[19] Webkit: Open source web browser engine. webkit.org.

[20] G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *Proc. 2010 Conference on Programming Language Design and Implementation*, PLDI '10, pages 174–186. ACM, 2010.

[21] G. Xu and A. Rountev. Precise memory leak detection for java software using container profiling. In *Proc. 30th International Conference on Software Engineering*, ICSE '08, pages 151–160. ACM, 2008.

[22] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *Proc. 2010 Conference on Programming Language Design and Implementation*, PLDI '10, pages 160–173. ACM, 2010.