Low-Overhead Memory Leak Detection Using Adaptive Statistical Profiling

Trishul M. Chilimbi
Microsoft Research
One Microsoft Way
Redmond, WA 98052
trishulc@microsoft.com

Matthias Hauswirth
Computer Science Dept.
University of Colorado at Boulder
Boulder, CO 80309-0430
matthias.hauswirth@colorado.edu

ABSTRACT

Sampling has been successfully used to identify performance optimization opportunities. We would like to apply similar techniques to check program correctness. Unfortunately, sampling provides poor coverage of infrequently executed code, where bugs often lurk. We describe an adaptive profiling scheme that addresses this by sampling executions of code segments at a rate inversely proportional to their execution frequency.

To validate our ideas, we have implemented SWAT, a novel memory leak detection tool. SWAT traces program allocations/frees to construct a heap model and uses our adaptive profiling infrastructure to monitor loads/stores to these objects with low overhead. SWAT reports 'stale' objects that have not been accessed for a 'long' time as leaks. This allows it to find all leaks that manifest during the current program execution. Since SWAT has low runtime overhead (< 5%), and low space overhead (< 10% in most cases and often less than 5%), it can be used to track leaks in production code that take days to manifest. In addition to identifying the allocations that leak memory, SWAT exposes where the program last accessed the leaked data, which facilitates debugging and fixing the leak. SWAT has been used by several product groups at Microsoft for the past 18 months and has proved effective at detecting leaks with a low false positive rate (<10%).

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification – *reliability, statistical methods*.

General Terms

Reliability, Verification, Performance, Measurement.

Keywords

Low-overhead monitoring, runtime analysis, memory leaks.

1. INTRODUCTION

The complexity of current software and machine architectures has made application profiling an important part of many performance analysis and optimization frameworks. Coarse-grain program profiles are easily obtained by tracing events of interest, while fine-grain profiles often require sampling techniques to reduce runtime overhead.

With the growing interest in program correctness, researchers have developed a variety of static analysis and runtime tools to detect program errors. Many tools for runtime program checking, such as Eraser (a data race detector) [19], and Purify (a memory leak tool) [10], require monitoring program events of interest at a very fine granularity. Consequently, these tools have high runtime overhead (5–30X), which limits their use.

We would like to use sampling to reduce the overhead of runtime program checking tools. Unfortunately, while sampling provides good coverage of events in frequently executed code segments, bugs often lurk in rarely executed portions of a program. We describe an adaptive profiling scheme that addresses this shortcoming by sampling executions of code segments at a rate inversely proportional to their execution frequency. Thus, rarely executed code segments are effectively traced whereas frequently executed code segments are sampled at a very low rate. This approach trades the ability to collect more samples from frequently executed code segments for more comprehensive code coverage, while maintaining similar runtime overhead.

We informally characterize the types of program bugs most suited to detection by a sampling approach using two criteria. If B is a program bug that is associated with program event E, then sampling event E is effective at detecting B if (1) when bug B occurs, then event E does not occur, ensuring that sampling does not produce any false negatives, and (2) if bug B is not present, then the number of occurrences of event E should exceed the reciprocal of the sampling rate, ensuring that the number of false positives reported is low.

To validate our ideas, we have implemented SWAT, a novel memory leak detection tool. Memory is leaked when an allocated object is not freed, but is never used again. SWAT predicts whether an object is still going to be accessed in the future given its past history of accesses. An object o, that is predicted not to be accessed anymore until the end of the program, is called stale and identified as a leak. SWAT traces program allocations/frees and samples data accesses to heap objects using our low-overhead adaptive profiling infrastructure. It reports 'stale' heap objects that have not been accessed for a (user definable) 'long' time as leaks. SWAT has three main advantages over the widely used Purify tool from Rational [10]. It uses a unique strategy of identifying leaks based on object staleness and is consequently able to find all leaks that Purify would report, plus additional leaks (the amount of false positives reported are low). In fact, SWAT's strategy is guaranteed to find all leaks that manifest during the given program execution. Next, since it uses sampling, the overhead is significantly lower than Purify (5% Vs. 3–5X). This makes it possible to use SWAT to track leaks in production code that take days to manifest. Finally, SWAT provides an

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ASPLOS'04, October 9–13, 2004, Boston, MA, USA.
Copyright 2004 ACM 1-58113-000-0/00/0004...$5.00.

indication of which program instruction last accessed the leaked object, which often enables faster debugging and leak fixing. SWAT has been used by several product groups at Microsoft for the past 18 months and has proved effective at detecting leaks with a low false positive rate (<10%).

The three main contributions of this paper are:

• design and implementation of a sample-based adaptive profiling scheme suitable for program checking. In addition, we informally characterize the type of program errors that can be effectively detected using sampling (Section 2).

• design and implementation of a novel memory leak detector, SWAT, that uses our adaptive profiling framework to achieve low overhead and has several advantages over state-of-the-art runtime leak detectors (Section 3).

• evaluation results, including real-use case studies, that demonstrate that SWAT is effective at finding memory leaks with a low false positive rate (< 10%), despite using a low enough sampling rate to incur a runtime overhead of around 5% (Section 4).

2. ADAPTIVE PROFILING

This section describes an implementation of adaptive profiling that extends the bursty tracing infrastructure described by Hirzel and Chilimbi [14]. Bursty tracing is a sampling infrastructure that can switch between no program monitoring and complete event tracing with very low overhead and is an extension of a sampling framework described by Arnold and Ryder [2]. The frequency of transitions between these two modes and the length of time spent in complete tracing is controlled by two user specified sampling counters. One of the drawbacks of bursty tracing is that its sampling methodology may miss infrequently executed code segments that are nevertheless important for identifying program errors. Adaptive bursty tracing addresses this shortcoming by starting out with full program tracing and adaptively tuning the sampling rate for individual code segments, such that frequently executed code regions are sampled at a progressively lower rate. We first provide a brief overview of bursty tracing and then describe adaptive bursty tracing.

2.1 Bursty Tracing

Bursty tracing is a sampling-based technology for low-overhead, continuous program monitoring. Unlike traditional sampling, bursty tracing can periodically capture complete program execution detail (i.e., a 'trace sample') for short timeframes. Two adjustable sampling counters control sample frequency—how often a trace sample is collected—and sample size—and how big it is. The primary advantage of bursty tracing over conventional sampling is the temporal execution detail it provides that can be invaluable for debugging and fixing detected program defects. Further, bursty tracing permits additional control and flexibility by allowing the trace sample extent to be adjusted in addition to collection frequency.

We provide a brief overview of bursty tracing, which is described in further detail elsewhere [14]. In the bursty tracing framework, the code of each procedure is duplicated (see Figure 1). Both versions of the code contain the original instructions, but only one version is instrumented to also profile events of interest. Both versions of the code periodically transfer control to dispatch checks at procedure entries or loop back-edges. The dispatch checks use a pair of counters, *nCheck* and *nInstr*, to decide in which version of the code execution should continue.

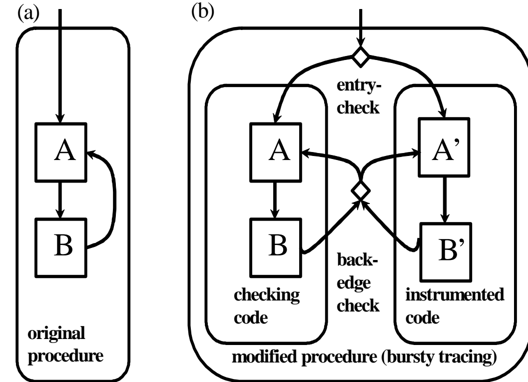

Figure 1. Instrumentation for low-overhead temporal profiling

At startup, *nCheck* is *nCheck0* and *nInstr* is zero. Most of the time, the checking code is executed, and *nCheck* is decremented at every check. When it reaches zero, *nInstr* is initialized with *nInstr0* (where $nInstr0 \ll nCheck0$) and the dispatch check transfers control to the instrumented code. While in the instrumented code, *nInstr* is decremented at every check. When it reaches zero, *nCheck* is initialized with *nCheck0* and control returns back to the checking code.

The bursty tracing profiling framework does not require operating system or hardware support and is deterministic. We implemented it using Vulcan [20], which is a binary rewriting system for the x86, and hence it does not require access to program source code or recompilation. The profiling overhead is easy to control: there is a basic overhead for the dispatch checks, and beyond that the overhead is proportional to the sampling rate $r = nInstr0/(nCheck0+nInstr0)$. Via *nCheck0* and *nInstr0*, we can freely chose the burst length and the sampling rate. In our experience, the basic runtime overhead of the dispatch checks is less than 5% and at a sampling rate of 0.1% the instrumentation overhead is negligible.

2.2 Adaptive Bursty Tracing (ABT)

While bursty tracing captures temporal execution detail of frequently executed code segments, many program defects only manifest on rarely visited code regions that sampling is likely to miss. Adaptive bursty tracing (ABT) addresses this shortcoming by sampling executions of code segments at a rate inversely proportional to their execution frequency. Thus rarely executed code segments are essentially traced whereas frequently executed code regions are sampled at a very low rate.

ABT maintains a per-dispatch check sampling rate rather than a global sampling rate for all dispatch check points. Initially all dispatch checks are sampled at a rate of 100% (i.e., full tracing). Each time a dispatch check is executed its sampling rate is reduced by an adjustable fractional amount (*Decr*) until a chosen lower bound sampling rate is reached (*Min*). The formula we use is: $nCheck0(n) = (Decr^{(n-1)} - 1)*nInstr0$

If $Decr = 10$, and $Min = 0.1\%$, this gives us sampling rates of 100% ($nCheck0(1) = 0$), 10% ($nCheck0(2) = 9*nInstr0$), 1% ($nCheck0(3) = 99*nInstr0$), and 0.1% ($nCheck0(4) = 999*nInstr0$) (as our sampling rate $r = (nInstr0)/(nCheck0+nInstr0)$). In the steady state, rarely executed code segments are virtually traced (sampled at close to 100%) while frequently executed code segments are sampled at the lower bound sampling rate. This approach trades the ability to collect more samples from frequently

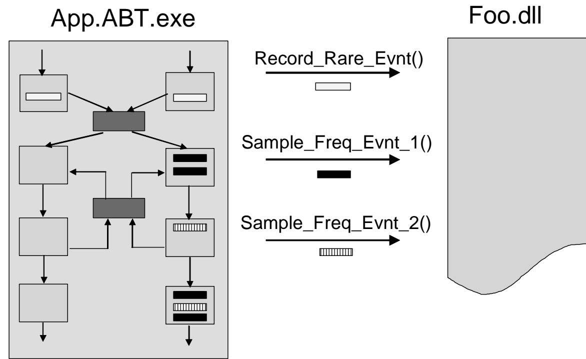

Figure 2. Adaptive Bursty Tracing Framework

executed code segments for more comprehensive code coverage. It maintains similar runtime overhead by keeping the total number of trace samples collected, roughly the same.

The counters *nCheck0* and *nInstr0* of the bursty tracing profiling framework (and *nCheck0*, *nInstr0*, *Decr*, *Min*, for the ABT framework) control its overhead and the amount of profiling information it generates. For example, setting *nCheck0* to 9900 and *nInstr0* to 100 results in a sampling rate of 100/10000=1% and a burst length of 100 dynamic checks.

The ABT framework is illustrated in Figure 2. Infrequent program events, such as dynamic heap allocations and lock acquisitions, are traced with conventional instrumentation (*Record_Rare_Evnt()*). Frequent events that are too expensive to trace, such as data references and branch executions, are monitored using ABT (*Sample_Freq_Evnt_i()*). All these events are communicated to a custom dll (Foo.dll) that either emits these to a log file for further processing or analyzes them online and reports errors.

2.3 Discussion

Sampling techniques, such as the ABT technique, are well-suited to detect many program errors, but not all. We can informally characterize the class of program bugs amenable to detection by a sampled approach with two criteria. If B is a bug that is associated with program event *E*, then sampling event *E* is effective at detecting *B* if:

Soundness condition:

$$B \Rightarrow \neg E$$

This condition says that if bug *B* occurs, then event *E* should not occur, ensuring that sampling does not produce any false negatives.

Preciseness condition:

$$\neg B \Rightarrow |E| > \frac{1}{(samplingrate)}$$

This states that if bug *B* is not present, then the number of occurrences of event *E* should exceed the reciprocal of the sampling rate, ensuring that the number of false positives reported is low.

Many liveness properties meet both conditions. Detecting uninitialized variables by sampling stores satisfies soundness but not preciseness. Detecting data races using Eraser's lockset algorithm and sampling shared accesses does not satisfy soundness and may or may not satisfy preciseness. Detecting buffer overruns by sampling and checking memory accesses satisfies neither condition.

3. SWAT: DETECTING MEMORY LEAKS

To validate our ideas, we have implemented SWAT, a novel memory leak detection tool that uses our ABT framework. We first provide some background information on leak detection and then describe SWAT.

3.1 Background

Memory is *leaked* when an allocated object is not freed, but is never used again. The central question for a memory leak detector is: Given a time *t* in the run of a program, and an object *o*, has *o* been *leaked*? It is generally impossible to determine, at time *t*, whether object *o* has been *leaked* or not. To answer this question, we need to know whether the object will be used in the future or freed before the program completes.

The question can only be definitely answered at the end of a program run. At that point, any object that has not been freed (immortal) is necessarily a leak. This approach has the drawback that objects often are intentionally not freed at the end of a program run, because the operating system will anyway reclaim the entire heap of the terminating process. Thus a leak detector based on immortal object identification will identify many uninteresting leaks. In addition, for server programs that need to run 24x7, objects that are not used but only freed at program termination should probably be classified as leaks.

It is also possible to give a conservative answer to the question at any point during program execution. One can distinguish between objects that are guaranteed to be *leaked*, and objects that might be *leaked*. Theoretically, the most precise conservative answer is to determine whether an object *o* is *dead* at time *t*. An object *o* is *dead*, if the program cannot reach any future state in which *o* is going to be accessed. However, an object that is not *dead*, may or may not be *leaked*. This is because for the object to be alive there only needs to be one possible future program path with an access, but the actual execution might take another path that does not access the object.

A practical, but less precise (it will find fewer leaks) approach to conservatively answer the question is to determine whether an object *o* is *unreachable* at time *t*. An object *o* is *unreachable*, if there is no reference chain to it from the root set (it is not reachable starting from global and stack variables). At time *t*, any *unreachable* object *o* necessarily is also *leaked* (and *dead*). But a reachable object may or may not be *leaked*.

Note that at a given time *t*, the set of *leaked* objects always includes the set of *dead* objects, and the set of *dead* objects always includes the set of *unreachable* objects.

The sample-based approach to memory leak detection described in this paper does not always provide the correct answer to the question whether object *o* is *leaked* at time *t*. It only provides an educated guess. But that guess can sometimes be even more useful than a correct conservative answer that often would be 'unknown'.

3.2 Overview

SWAT uses the following simple insight to detect memory leaks: ***if a heap object has not been accessed for a 'long' time, then it is a memory leak***. This simple invariant ensures that SWAT detects all leaks that manifest during the given runtime execution. However,

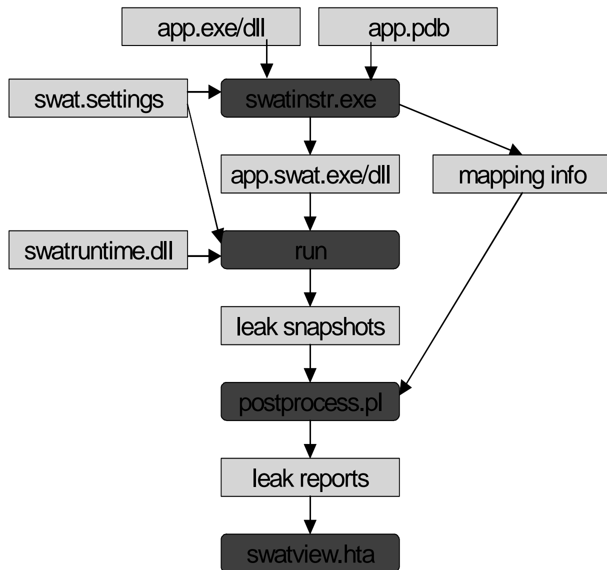

Figure 3. SWAT Infrastructure

there are two significant obstacles to implementing this staleness policy within a practical memory leak tool. First, the overhead of monitoring all heap data accesses can be prohibitive. Next, the leaks reported could include a large number of false positives. Perhaps, for these reasons, no existing memory leak tool uses this 'staleness' approach.

SWAT uses our adaptive profiling infrastructure to monitor heap accesses with low overhead. It uses a sampling rate of 0.1%, which entails a runtime overhead of less than 5% on average. Regarding false positives, our experience with SWAT indicates that tuning the 'time elapsed[1]' before an unaccessed heap object is reported as a leak is sufficient. In addition, many of the remaining false positives are of interest to developers since objects that have not been accessed for a very long time often indicate inefficient use of memory. Sampling the heap data accesses appears to have no noticeable impact on the number of false positives. This is because most heap objects are accessed multiple times and the sampling will have to miss **all** of these accesses for an active object to be mistakenly classified as a leak. We present supporting empirical evidence later.

In more detail, SWAT operates as shown in Figure 3. swatinstr.exe creates an instrumented version of the application, which is used in place of the original. A mapping file is also created to help associate program counter values with source code in the original copy. The instrumented application communicates all heap allocations/frees (*Record_Alloc()*) and a sampled set of heap accesses (*Sample_Load/Store()*) obtained via our adaptive profiling framework to a runtime dynamically loaded library (swat.dll) as shown in Figure 4. The runtime DLL uses the heap allocation/free information to maintain a model of the heap, which it updates with the heap access information. Periodically, swat.dll takes a snapshot, where it visits all objects in its heap models and reports all objects that satisfy its staleness predicate as leaks. It is able to associate the responsible heap allocation, all heap frees that freed objects created at that allocation site, and most importantly the 'last access', with each heap object reported as a leak. Our

[1] time elapsed is measured in number of heap accesses.

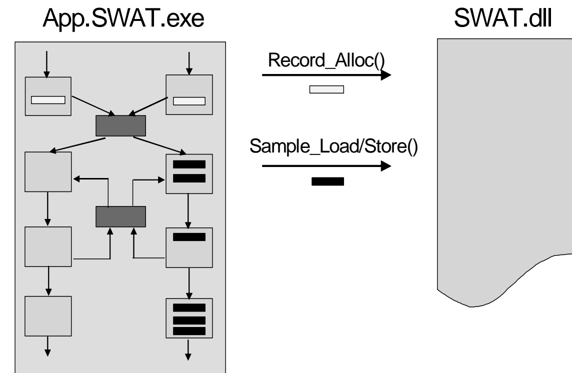

Figure 4. SWAT Instrumentation

experience indicates that this last access information is invaluable for quickly debugging and fixing detected leaks. In addition, the last access information enables quick determination of false positives. The leak snapshots are postprocessed and visualized through a GUI. The GUI includes a source code browser that highlights the last access to a leaked object.

3.3 Heap Model

The purpose of the heap model is to keep track of information about all allocated objects. The information maintained for an object is called an object descriptor. An object descriptor contains the object's allocation site, last access time, and last access site. The heap model has to implement the following interface in order to maintain and provide access to this information:

AllocateObject(ip, startAddress, size)
FreeObject(ip, startAddress)
FindObject(ip, address)
GetObjectIterator()

The instrumentations invoke the first two functions whenever an object is allocated or freed. Whenever AllocateObject is called, the heap model stores the allocation information in the object descriptor. The instrumentations of all the memory accessing instructions invoke the FindObject function. The address they provide to that function does not necessarily have to be the object's start address, but may point anywhere inside the object. The heap model updates the last access site (ip) and time each time FindObject is called. Since this function is invoked frequently, it is crucial for it to have minimal overhead. Finally, the GetObjectIterator provides a means to iterate over all currently allocated objects, retrieving all the available information for each object.

Various implementations of this interface are possible, but most of them are not practical. The most straightforward would be an inline-implementation, where the object descriptor is maintained in the object's header. A key issue with this approach would be how to find the object's start address, given a pointer pointing inside the object. Another approach would be maintaining a hash table of each possible address to the descriptor of the object spanning the given address. This table would have one entry for each allocated object, a considerable space overhead. A more advanced approach would be to keep such a hash table for only the start addresses of each object, maintaining a separate compact data structure that maps from any address to the start address. Since many objects are small, such a structure could encode the difference from an address to the corresponding object's start address in one or two bytes.

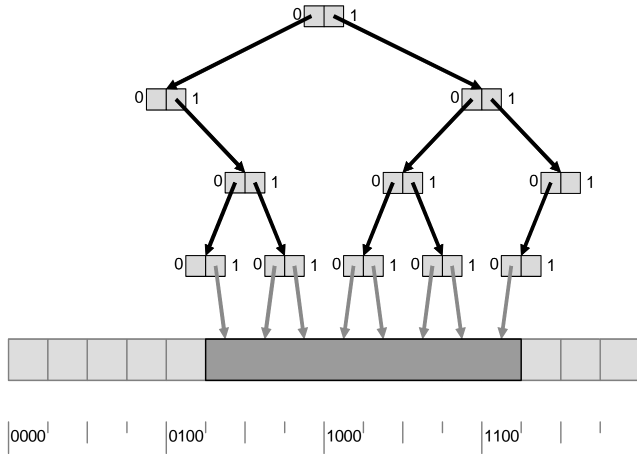

Figure 5. Part of complete address tree for object with start address 0101 and size 8.

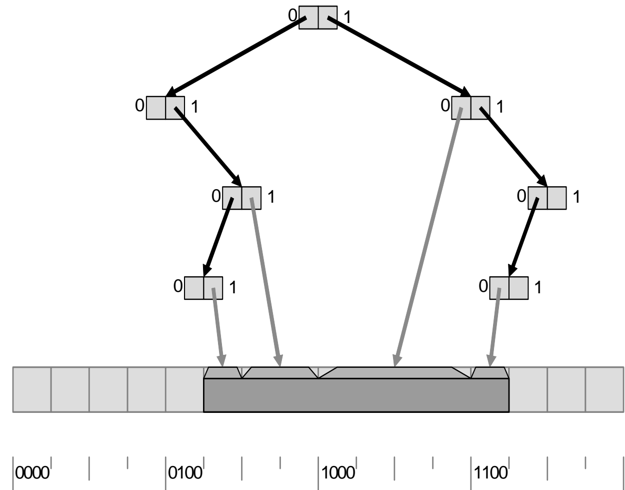

Figure 6. Part of an optimized address tree for an object with start address 0101 and size 8. All four unnecessary nodes are removed, and interior nodes directly reference the object descriptor.

Our concrete implementation of the heap model uses a different approach. We maintain a binary tree structure called an address tree (see Figure 5). Each level in this tree of depth 32 represents one bit in the 32 bits of an address. The root node represents the most significant bit. Its two children represent the two sets of addresses where the first bit is 0 or 1 respectively. To find the descriptor for an object spanning a given address, the FindObject function starts at the root of the tree and follows the path to the leaf describing the given address.

Maintaining a complete binary tree with a leaf node for every used heap address would require a large amount of memory. To reduce this space overhead we allow an interior node whose addresses all reference the same object to directly point to the descriptor of that object (see Figure 6). This eliminates a large amount of leaf nodes and their ancestors. The space savings are particularly big for large and for well aligned objects. Since allocators often align objects to 4 byte boundaries (and thus objects' start addresses and sizes often are a multiple of 4) we essentially save all nodes on the lowest two layers. But in contrast to other approaches that might exploit the 4 byte object alignment, our approach is flexible and still works in the presence of unaligned objects.

The time overhead of the FindObject function is constant, given a 32 bit address and the resulting 32 bit address tree. In the worst case we have to traverse 32 address tree nodes to reach the object descriptor. Note that the top level of the address tree, and the most frequently traversed address tree nodes are likely to reside in the data cache, which keeps memory access costs down. In addition, we batch and present tree operations in address sorted order to exploit locality. Finally, we use a low sampling rate of 0.1% for our experiments, which mitigates the impact of the FindObject operation on overall program execution time.

3.4 Staleness Predicates

Our leak detector periodically traverses the heap model, trying to identify leaked objects. The exact interval between traversals is user-configurable. While not currently implemented, interactive applications would probably benefit from scheduling these traversals when the program is idle waiting for user input. In addition, a leak report is always generated at application shutdown. We use accesses rather than wall clock time to measure staleness. This avoids labelling objects of an interactive application that is left idle overnight as stale. As described in the previous subsection, the heap model maintains information in the descriptor of each heap object. Given this information, a staleness predicate guesses whether the object is leaked or not. Figure 7 gives an overview of three different staleness predicates.

The *Never Accessed* staleness predicate considers every object that has never been accessed a leak. The *Constant Time* staleness predicate bases its guess on the length of time (measured in terms of observed data accesses) since the last access of an object. If that time is above a constant threshold the object is considered to be leaked. The *Active Time* predicate is similar to the constant time predicate. The difference is that the threshold is not a constant, but n times the active time of the object. The active time of an object is the time between its first and its last access so far. The intuition behind this predicate is that an object that has been active for a long time is allowed to be inactive for a long time, before it is considered a leak.

3.5 Leak Reporting

A central question for any practical memory leak detector is what to report to a user. The detector identifies leaked objects but these

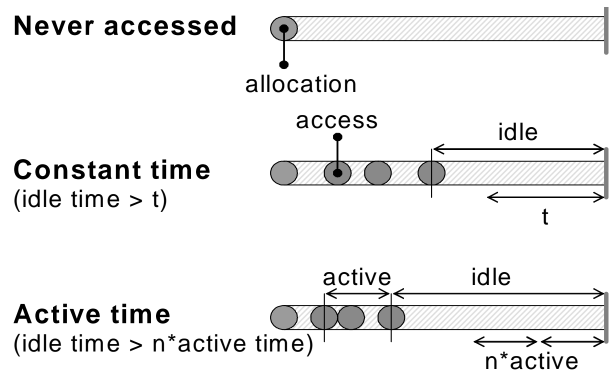

Figure 7. Staleness Predicates.

are caused by missing deallocation sites in the program. Since these deallocation sites are absent, the detector cannot pinpoint source code locations responsible for the leaked objects. Consequently, most leak detectors only report the allocation site of a leaked object. Insure++ also reports the point where the leaked object became unreachable [17]. Our statistical leak detector provides even more aggressive information with respect to where an object was leaked. Since we sample object accesses and keep track of the last access for staleness prediction, we are able to report the last observed access site in addition to the allocation site for each leaked object. Further, we report all deallocation sites of objects that were allocated at the same site as the leaked object, if any exist. This can further narrow down the region of code where the leaked object should have been deallocated.

3.6 Ordering Leaks

Our leak detector is not always correct. Sometimes it reports an object as leaked even though it is still going to be accessed in the future. In addition, it aggressively classifies unused objects as leaked (unlike previous detectors that only classify unreachable or immortal objects as leaked). This leads to the problem of false positives, which we address by ordering the list of reported leaks.

Leaks are identified at the level of individual objects with which we associate a last access site. They are grouped by allocation site. We support three different leak list sorting orders—by number of objects, by number of bytes, and by drag. An allocation site that leaks many objects is likelier to include real leaks. Ordering by number of leaked bytes ranks the more problematic leaks higher. Finally, the true cost of a leak is not just how many bytes it leaked, but the space-time product of leaked bytes multiplied by the time the leaked objects used up space. This product, which represents the opportunity cost of not reusing the leaked memory, is called the leak's drag. Ordering the leak list by drag ranks the most expensive leaks at the top. Our experience indicates that first sorting the allocation sites by number of leaked objects and then focusing on high ranking leaks with the largest drag yields the best results.

3.7 Discussion

While collecting bursts rather than point samples provides little advantage for detecting leaks, it provides contextual information that can be useful while debugging leaks. SWAT has an option that records program burst history at the last access site for leaked objects.

The adaptive tracking of rarely executed code segments is critical to SWAT. Without this, data objects that are not accessed in frequently executed code segments would be erroneously reported as leaks. Our first attempt at a staleness-based leak predictor used bursty tracing and suffered from a large number of false positives.

4. EXPERIMENTS

We evaluated SWAT across several dimensions. We performed four sets of experiments to evaluate SWAT's runtime and space overhead, the impact of sampling on leak detection, and to evaluate our staleness predicates. Finally we present case studies that indicate that SWAT is effective at detecting memory leaks with low false positives.

4.1 Benchmarks

We used the SPECInt2000 benchmarks for our experiments. Measurements were performed by running the benchmarks on one processor of a dual processor 2.7 GHz Pentium 4 PC with 2 GB RAM running Windows XP. The SPEC benchmarks were run with their largest data set (ref). All timing runs were performed five times and the results were averaged. The observed variation across runs was less than 2% unless otherwise noted. We performed three sets of experiments to evaluate SWAT's runtime overhead, the impact of sampling on leak detection, and to evaluate our staleness predicates.

4.1.1 Runtime Overhead

We measured the runtime overhead of SWAT using our adaptive profiling scheme with a *Decr* value of 10 and *Min* values set to 0.01% and 0.1% respectively. *nInstr0* was set to 10 to produce bursts of length 10. The results are shown in Figure 8. In both cases the runtime overhead was less than 5% for most of the benchmarks. For these runs SWAT was configured to produce only produced one leak report at the end, which was not included in the execution time. As a result, the numbers reflect only the access monitoring and heap model construction time.

4.1.2 Space Overhead

Figure 9 shows the space overhead incurred by running SWAT. The data was obtained by taking multiple heap snapshots during program execution and computing their average. Part of this overhead is attributable to the address tree data structure used to model the heap and the rest to the data associated with each heap object to identify leaked objects as well as their last access site.

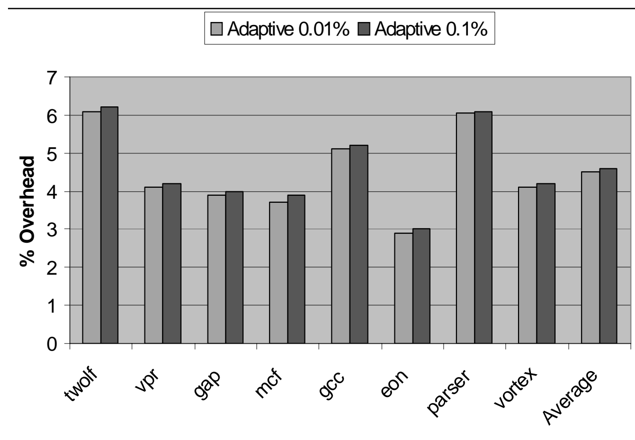

Figure 8. Runtime Overhead of SWAT.

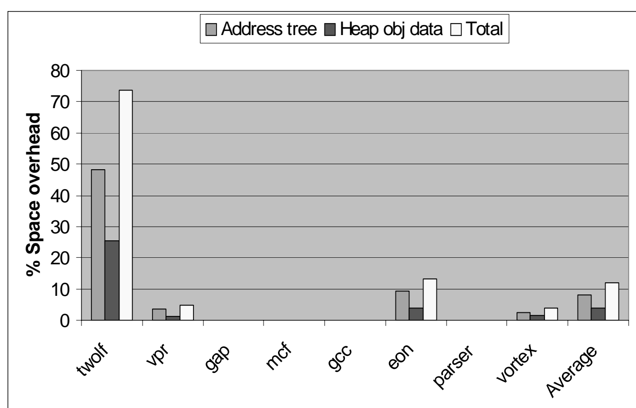

Figure 9. Space overhead of SWAT.

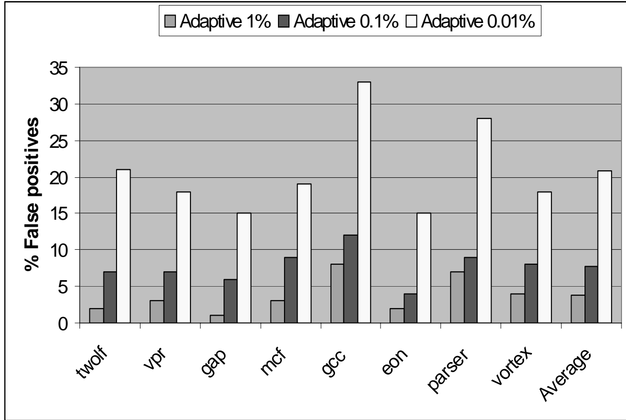

Figure 10. False positives introduced by sampling.

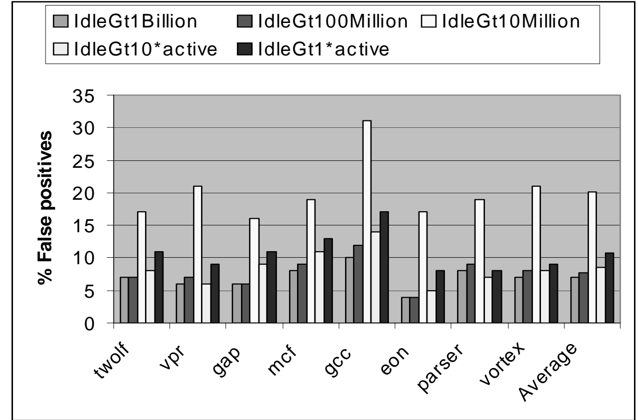

Figure 11. False positives introduced by various staleness predicates.

With the sole exception of twolf, which has an extremely large number of very small objects, the total space overhead is quite small, averaging less than 10%. For many of the benchmarks (gap, mcf, gcc, parser) the space overhead is lower than 0.1%.

4.1.3 Impact of Adaptive Profiling on Leak Detection

We ran the following experiment to evaluate false positives introduced by sampling. First, we injected leaks into the benchmarks by randomly removing 10% of all dynamic deallocations. Next, we set the sampling rate to 100% and counted the leaks that were detected using an *IdleGt100Million* references staleness predicate. Then we measured the additional leaks reported (false positives) at different adaptive sampling rates. The results are shown in Figure 10. For sampling rates of 1% and 0.1% the false positive rate is reasonably low. This is not completely surprising given our observation in Section 2.3 that sampling is well suited for errors that are associated with events that occur frequently enough to be detected despite sampling. So as long as non-leaked objects are accessed more often than 100 and 1000 times respectively, these are unlikely to be reported as false positives.

4.1.4 Staleness Predicate Evaluation

To evaluate the different staleness predicates, we used the same methodology of randomly removing 10% of all dynamic deallocations and comparing the false positive rate against the leaks reported by full tracing. The results are shown in Figure 11. While the *IdleGt1Billion* predicate performs well, it may miss leaks in programs that do not run for a long enough time. The *IdleGt100Million* predicate performs almost as well and the *IdleGt10*active* is comparable being slightly worse in most cases but better in a few.

4.2 Case Studies

SWAT has been used by several product groups at Microsoft for the past 18 months and has been effective at finding leaks with a low false positive rate. We report on four representative case studies—a large, interactive web software application, a multimedia application, and two PC games.

4.2.1 Large Interactive Web Application

First, to test SWAT, we artificially injected 34 leaks in a large interactive web application by randomly skipping frees at runtime and ran the app for a couple of hours. SWAT correctly identified all 34 injected leaks with only one 'potential' false positive (this turned out to be a real leak). Next, we applied SWAT to the original application and gave this version to members of our research group to use on a daily basis as a replacement. Since we lacked a mechanism to capture and replay external inputs, we were unable to report the runtime overhead incurred. However, the overhead of running this version of the application was sufficiently low that it was indistinguishable from the original version. In addition, as reported in Table 1 the space overhead was extremely low. To date, over a period of 6 months, SWAT has detected around 20 'potential' leaks ('potential' as they have not been verified yet by the developers).

4.2.2 3rd Party PC Game (Strategy)

We were contacted a couple of months prior to its ship date and asked to run SWAT to identify memory leaks. The overhead of the SWAT instrumented version of the game was sufficiently low that it was playable (< 5%). Since state-of-the-art PC games are very

	% Runtime Overhead		% Space Overhead		
Benchmark	**Adaptive 0.01%**	**Adaptive 0.1%**	**Address tree**	**Heap object data**	**Total**
Interactive Web App	NA	NA	0.18	0.27	0.45
PC Game (Strategy)	4.3	4.71	8.2	2.21	10.41
Multimedia App	4.78	5.23	0.0	0.13	0.13
PC Game (Simulation)	4.42	4.97	3.25	0.78	4.03

Table 1: SWAT overhead for real-world applications.

resource intensive and push the limits of desktop performance, this was a very encouraging result. The total space overhead was around 10%. The leak report was generated from around two hours of actual gameplay. SWAT reported several memory leaks. While many were 'false positives[1]', these corresponded to heap objects that had not been accessed for a very long time, suggesting memory inefficiency. For example, cached objects that were never subsequently accessed. After looking at the SWAT leak report, the lead developer decide to rewrite the memory management routines to track heap allocation better, despite being so close to the ship date.

4.2.3 Multimedia Application

We applied SWAT to the latest beta version of a multimedia application, one month before its scheduled release. As before, the SWAT instrumented version of the application was indistinguishable from the original version permitting long testing scenarios that reflected actual use. SWAT detected 6 leaks in the visual browser component of the application with no false positives. These were reported to the developers and fixed prior to shipping.

4.2.4 First Party PC Game (Simulation)

We were asked to run SWAT as part of a code quality review. The test scenario involved almost two hours of actual gameplay. We captured and replayed the scenario to measure the runtime overhead incurred and report this in Table 1. Both runtime and space overhead were less than 5%. SWAT detected 5 'potential' leaks (not yet verified by developers). We manually inspected the relevant code and believe that these include no false positives.

4.2.5 Discussion

In all cases the overhead of running the SWAT instrumented binary was low enough to permit monitoring real usage rather than using short, contrived scenarios. Our experience with SWAT suggests that it can be viably run all the time. Sampling is often criticized for missing information. However, we believe such objections are shortsighted as the total information gathered is the product of sampling rate and the number of program instructions monitored. Since lower sampling rates imply lower overhead, they encourage more people to run monitored applications longer, possibly producing more information overall.

The 'last access' information included in the leak reports was invaluable for fixing leaks as well as separating leaks from false positives. For three of the applications, the number of false positives was very low. However, as the case of the PC game (Strategy) indicates, even false positives can be interesting. In this case the SWAT data was used to re-architect and rewrite the memory management routines rather than fix the leaks.

5. RELATED WORK

The two closest areas of related work are bursty tracing and memory leak detection.

5.1 Bursty Tracing

The idea of reducing the cost of potentially high-overhead instrumentations by sampling was introduced by Arnold and Ryder [2]. Chilimbi and Hirzel [14] improved upon this infrastructure to allow the sampling of bursts of instrumentations, allowing for temporal profiles. They applied their bursty tracing framework to dynamic detection and prefetching of hot data streams [6]. We extend their framework by dynamically increasing the coverage of instrumentation in rarely executed code, while still achieving low overhead.

Liblit et al. [15] extend the Arnold-Ryder framework in a different dimension. Instead of uniformly spacing samples, they introduce a mechanism to trigger samples based on a geometric distribution. Their resulting data is a statistically rigorous fair random sample, and thus amenable to a large domain of statistical analyses. They too apply their framework to detect program errors. However, their focus is on detecting the cause of errors from multiple runs of a program, while our focus is on low-overhead program checkers that detect specific types of bugs on individual program runs.

5.2 Memory Leak Detection

There exist three approaches to memory leak detection. First, static analysis tools [11, 5] provide a way to identify leaking sites before actually running the program. They can find leaks before the program is deployed and do not cause any runtime overhead, but their lack of dynamic information leads to conservative results that include false positives and they cannot find all possible leaks.

Second, some dynamic tools [4,8,16,18] take, compare and inspect snapshots of the heap and allow the interactive analysis of memory consumption. They provide statistics about the amount of memory allocated at various sites, and for various types or object sizes, and they visualize the connectivity of heap objects, supporting the search for the causes of leaked objects, particularly in runtime environments with a garbage collector.

And third, automatic dynamic tools provide a list of leaked objects and leak sites at the end of the program run. Some of those tools [1,21] only capture immortal objects. Thus they only need to instrument the allocation functions. Others [10,9,7] capture unreachable objects. They do this by instrumenting the allocation functions, and by using a mark and sweep garbage collection approach [3] to garbage detection. Insure++ [17] uses runtime pointer tracking, a reference counting garbage collection approach, to detect the point at which the last reference to an object disappears. These tools either miss many leaks and/or incur high overhead, which limits their applicability.

As Hirzel et al. [12, 13] show, the quality of the results of a reachability-based leak detector largely depends on the type and liveness accuracy of its reachability traversal. They find that the liveness accuracy is more important. Our approach circumvents the problem of a precise liveness analysis by dynamically assessing the liveness of objects by observing accesses to those objects. We are not aware of any other automatic dynamic memory leak detection tool that captures leaks based on object accesses. Our approach can potentially capture leaked objects that are still reachable, and it does that with low enough overhead to be used in a deployed system.

6. FUTURE WORK

We plan on exploring applications of our adaptive profiling framework to other program bugs. For some types of bugs, it may only involve implementing a well known solution within our adaptive profiling framework, to take advantage of the low overhead monitoring capability. In other cases, such as SWAT, it may involve attacking a problem with a completely new approach that our adaptive profiling framework has made possible.

[1] Many of the false positives would be considered real leaks if the game is played only once, rather than multiple times in succession without shutdown. In the test scenario, the game was played just once.

With respect to SWAT, we would like to evaluate the impact of using an imperfect but more compact heap model. For programs with many small objects, our current heap model still contains a considerable number of address tree nodes. We could use a compact encoding to map from an arbitrary address to the start address of the object spanning that address, giving up precision where that offset cannot be stored in the small constant amount of space (e.g. a fraction of a byte) available per object.

Further, it would be interesting to compare our adaptive bursty tracing approach to Liblit's random sampling approach [15] within our leak detector.

It would be interesting to empirically compare the usability of our approach of identifying the site the leaked object was last accessed to the more conservative Insure++ [17] approach of identifying the site the object became unreachable. Our approach can identify the earliest possible location where the object could be deallocated (right after accessing it for the last time) for the given run. The Insure++ approach identifies the latest possible location where the object could be deallocated (right before loosing the last reference to it). It would be interesting to combine both approaches, so, for each leak, the user could see the allocation site, the site of the last access, and the site the object became unreachable. Further, the user could see all deallocation sites of other objects allocated at the same allocation site. An extension to this would be to produce a trace of the program path taken between the last access site and the site the object became unreachable, giving the user the complete set of program points where the deallocation could be placed to prevent the leak.

7. CONCLUSIONS

We have described an adaptive profiling framework for implementing low-overhead runtime program checking tools. With sufficiently low-overhead, such program checkers could ship with production code and detect errors that occur during real use. We informally characterize the types of program errors that can be effectively detected using our framework. To validate our ideas, we have implemented SWAT, a novel memory leak detection tool. SWAT has been used by several product groups at Microsoft for the past 18 months and has proved effective at detecting leaks with a low false positive rate.

8. ACKNOWLEDGMENTS

We would like to thank Andrew Edwards, Bill Gates, Jim Larus, Jon Pincus, Amitabh Srivastava, Ben Zorn, and our anonymous referees for several helpful comments.

9. REFERENCES

[1] Erwin Andreasen and Henner Zeller. LeakTracer. August 2003. http://www.andreasen.org/LeakTracer/

[2] Matthew Arnold and Barbara Ryder. A framework for reducing the cost of instrumented code. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI),* 2001.

[3] Hans Boehm, Alan Demers, and Mark Weiser. A garbage collector for C and C++. http://www.hpl.hp.com/personal/Hans_Boehm/gc/

[4] Borland OptimizeIt Profiler. http://www.borland.com/optimizeit/optimizeit_profiler/index.html

[5] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. In *Software Practice and Experience,* 2000; 30:775-802.

[6] Trishul M. Chilimbi and Martin Hirzel. Dynamic Hot Data Stream Prefetching for General-Purpose Programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 199-209, June 2002.

[7] Jeremy Dion and Louis Monier. ThirdDegree. http://research.compaq.com/wrl/projects/om/third.html

[8] EJ-technologies' JProfiler. http://www.ej-technologies.com/products/jprofiler/overview.html

[9] GreatCircle. http://www.geodesic.com/

[10] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, pages 125-136, 1992.

[11] David L. Heine and Monica S. Lam. A Practical Flow-Sensitive and Context-Sensitive C and C++ Memory Leak Detector. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI),* 2003.

[12] Martin Hirzel, Amer Diwan, and Antony Hosking. On the Usefulness of Liveness for Garbage Collection and Leak Detection. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 181-206, June 2001.

[13] Martin Hirzel and Amer Diwan. On the Type Accuracy of Garbage Collection. In *International Symposium on Memory Management (ISMM)*, pages 1-11, October 2000.

[14] Martin Hirzel and Trishul M. Chilimbi. Bursty Tracing: A Framework for Low-Overhead Temporal Profiling. In *4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO)*, pages 117-126, December 2001.

[15] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug Isolation via Remote Program Sampling. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2003.

[16] Paul Moeller. Win32 Java Heap Inspector. 1998. http://www.geocities.com/moellep/debug/HeapInspector.html

[17] Parasoft Insure++. http://www.parasoft.com/products/insure/

[18] Quest jProbe Memory Debugger. http://www.quest.com/jprobe/debugger.asp

[19] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Race Detector for Multi-Threaded Programs. In *ACM Transactions on Computer Systems*, 15(4): 391-411, November 1997.

[20] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. In *Microsoft Research Tech Report, MSR-TR-2001-50*, 2001.

[21] Benjamin Zorn and Paul Hilfinger. A memory allocation profiler for C and Lisp programs. In *Proceedings of the Summer USENIX Conference*, pages 223-237, 1988.