

Low-Overhead Run-Time Memory Leak Detection and Recovery

Timothy Tsai, Kalyan Vaidyanathan, and Kenny Gross
Sun Microsystems, Inc.
{timothy.tsai,kalyan.vaidyanathan,kenny.gross}@sun.com

Abstract

Memory leaks are known to be a major cause of reliability and performance issues in software. This paper describes a run-time scheme that detects and removes memory leaks with minimal performance overhead and with no modifications to application source code. The scheme consists of a first stage where a pattern recognition technique proactively detects subtle memory leaks, followed by a more resource-intensive second stage that scans the memory space of an application and removes detected memory leaks. The pattern recognition technique in the first stage is based on the multivariate state estimation technique (MSET) which provides accurate detection of subtle memory leaks with very little overhead. The second stage is only activated when problems are identified by the first stage. For our prototype, this second stage is based on debugging and analysis tools provided by Solaris 10. Due to the low-overhead impact of the first stage, the system can be monitored for memory leaks without incurring noticeable performance degradation. We present and discuss some results from our unique proactive detection and debugging methodology.

1 Introduction

Memory leaks are known to be a major cause for unreliable and poorly performing software. A memory leak is a software error in which dynamic memory is not freed after its useful lifetime. Although memory leaks strictly refer to any existence of non-useful dynamic memory that has not been deallocated, most often memory leaks are associated with a programming error in which the a program loses even the ability to release the non-useful memory. For example, an error might overwrite a pointer to a memory area thus rendering the memory unreachable and preventing the program from either utilizing the memory or freeing it.

Although the decreasing cost of memory has produced systems with increasing amounts of physical memory, memory leaks can still result in a negative impact on performance and reliability, especially in a

multi-user environment. A long-running program or server program that must be continuously available often encounters the programming errors responsible for generating memory leaks on an ongoing basis, resulting in an ever increasing total memory usage for that program. Eventually the amount of free physical memory decreases to a point where a new request for memory, either by the same program or a different program, cannot be satisfied by the operating system. The system's inability to satisfy the request for memory usually leads to the failure of the requesting program or even to a system panic if the operating system itself is the requester. If the operating system supports virtual memory and swap space, a portion of virtual memory can be swapped out to free up a corresponding amount of physical memory. Unfortunately, if the swapped out memory is still actively used by another program, a performance overhead is incurred to swap that memory back in, which sometimes leads to a thrashing situation where virtual memory is continuously swapped in and out. Also, swap space is finite in size and is ultimately exhausted in the face of ongoing memory leakage, leading to application or system failure.

Because memory leaks can have such a negative impact on performance and reliability, many tools exist to detect the programming errors that cause memory leaks. Some tools analyze source code to detect programming constructs that lead to memory leaks [1]. Other tools instrument either source code or binary code to insert additional debugging support that not only detects memory leaks but also many other types of bugs associated with dynamic memory usage and management [2][3][4][5][6][7][8]. Many of these tools are quite advanced, and many are offered commercially. These tools are quite effective because they generally detect all accesses to dynamic memory. The trade-off is the incurring of the large performance overhead associated with such debugging support. Still, because these tools are effective in finding bugs, they are widely used.

Some tools have taken the idea of memory leak detection beyond the testing lab by performing detection on deployed systems [9]. Memory leak detection and removal for run-time systems is often called garbage collection. Boehm [10][11] has

proposed garbage collection techniques suitable for run-time usage, including conservative garbage collection techniques that minimize false positive identification of memory that is no longer used. Conservative garbage collection techniques usually follow either a *reference-count* or a *mark-sweep* strategy. Both strategies attempt to identify objects that are still live or in use by the application. The reference-count strategy maintains a current count of the references to each object by incrementing the count whenever a pointer is set to refer to the object and decrementing the count whenever a reference is deleted. When the count for a particular object reaches zero, the memory for that object can be freed. The implementation of a reference-count strategy requires compiler or run-time environment support for maintenance of reference counts. The mark-sweep strategy attempts to accomplish the same goal without requiring current counts of references to objects. Instead, an on-demand traversal of memory is used to mark objects that are currently the target of at least one reference. Upon completion of the traversal, all objects that have not been marked as a reference target are designated for deallocation. Our prototype uses a conservative garbage collection algorithm that is based on the mark-sweep approach.

One of the significant challenges for garbage collection is the additional performance overhead incurred. This overhead is particularly conspicuous for the mark-sweep approaches because they require a temporary suspension of the application while the algorithm is executed. This paper proposes some techniques to address these performance concerns in two ways: (1) We utilize an advanced pattern recognition technique to minimize the need for suspension of the application and execution of the mark-sweep algorithm, and (2) our implementation of the mark-sweep algorithm permits off-line execution and thus allows the application to resume execution before the entire algorithm is finished.

The remainder of this paper is organized as follows. Section 2 describes the software architecture of the complete scheme, including Stage 1 components for proactive identification of systems that exhibit memory leaks and Stage 2 components for memory leak detection, recovery, and fault injection. In Section 3, we present experiments using a prototype to evaluate the associated overheads associated and

the effectiveness of the prototype's memory leak detection algorithm. Finally, we present conclusions and ideas for future work in Section 4.

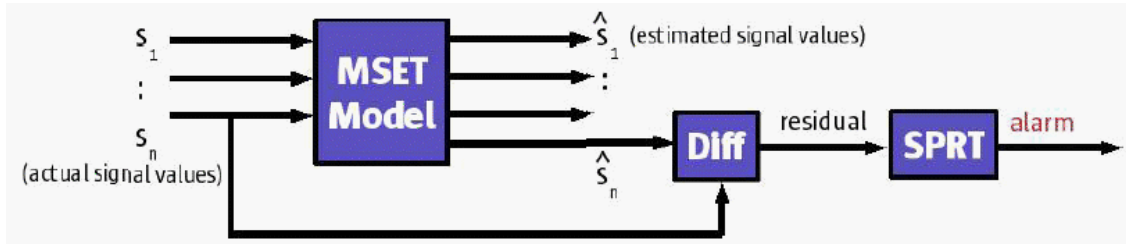
2 Description of Prototype

The key contribution of this paper is the combination of two memory leak detection stages to both minimize overhead and effectively detect and remove memory leaks. Stage 1 is a statistical pattern recognition technique that can raise alarms for memory leak situations at a system level as well as at an application level. Because Stage 1 incurs virtually no overhead, continuous monitoring of an entire system is practical. When Stage 1 raises an alarm, Stage 2 is invoked to perform low-level, conservative, mark-sweep memory leak detection and removal.

2.1 Stage 1: Pattern Recognition Technique to Trigger Garbage Collection

The multivariate state estimate technique (MSET) [12][13][14] is a non-linear, non-parametric modeling method that was originally developed by Argonne National Lab (ANL) for high-sensitivity proactive fault monitoring applications in commercial nuclear power applications where plant downtime can cost utilities and their constituents on the order of one million dollars a day. MSET techniques have been successfully applied in a number of reliability-critical applications [15][16], including monitoring of NASA Space Shuttle's main launch vehicle engine sensors, military gas turbine engines, industrial process equipment, high-performance computers, commercial jet engines, and nuclear power plant sensors.

In our paper, MSET refers to generic non-linear, no-parametric regression and not to any commercial implementation. A block diagram of MSET operation is shown in Figure 1. The MSET framework consists of a training phase and a monitoring phase. The training procedure is used to characterize the monitored equipment using historical, error-free operating data covering the envelope of possible operating regimes for the system variables under surveillance. This training procedure processes and evaluates the available training data S_1, S_2, \dots, S_n in Figure 1, then selects a subset of the data observations



that are determined to best characterize the monitored asset's normal operation. It creates a stored model of the equipment based on the relationships among the various signals. Some degree of correlation, linear or non-linear, among the signals is necessary for the model. This model is then used in the monitoring procedure to estimate the expected values of the signals under surveillance.

In the monitoring phase, new observations for all the system signals are first acquired. These observations are then used in conjunction with the previously trained MSET model to estimate the expected values of the signals $\hat{S}_1, \hat{S}_2, \dots, \hat{S}_n$ in Figure 1. MSET estimates are typically extremely accurate, with error rates that are usually only 1 to 2 percent of the standard deviation of the input signal.

The difference between a signal's predicted value and its directly sensed value is termed a residual. The residuals for each monitored signal are used as an anomaly indicator for sensor and equipment faults. Instead of using simple thresholds to detect fault indications, MSET's fault detection procedure employs a SPRT (sequential probability ratio test) [17] to determine whether the residual error value is uncharacteristic of the learned process model and thereby indicative of a sensor or equipment fault. The SPRT algorithm is a significant improvement over conventional threshold detection processes in that it provides more definitive information about signal validity with a quantitative confidence factor through the use of statistical hypothesis testing. This approach allows the user to specify false alarm and missed alarm probabilities, allowing control over the likelihood of false alarms or missed detection. This is a superior surveillance approach since the SPRT is sensitive not only to disturbances in the signal mean, but also to very subtle changes in the statistical quality (variance, skewness, bias) of the signals. For sudden, gross failures of a sensor or component under surveillance, the SPRT procedure announces the disturbance as fast as a conventional threshold limit check. However, for slow degradation, this procedure can detect the incipience or onset of the disturbance long before it would be apparent with conventional threshold limits.

2.2 Stage 2: Off-line Garbage Collection

We utilize a mark-sweep strategy for garbage collection that is based on existing utilities offered by the Solaris operating system (i.e., libumem.so [18], gcORE, and mdb [19]) that facilitate memory leak detection. Experimental results are discussed in Section 3.

The memory leak detection in our prototype is based on three Solaris software components: libumem.so, gcORE, and mdb. Libumem.so is a user-space slab allocator [20][21] that manages user-space

dynamic memory in a very efficient manner. More importantly for our work, libumem.so also provides debugging support that is useful for detecting memory leaks. This debugging support inserts additional metadata associated with each memory allocation. The gcORE utility creates a core image file for a process. When used for a process loaded with libumem.so, the core image file also contains the debugging metadata. This core file is then analyzed by the mdb debugger, a flexible debugger that supports both kernel and user process debugging with a large assortment of modules. Of particular interest is the `::findleaks` module which analyzes the core file and identifies memory leaks, using a conservative mark-sweep algorithm.

Figure 2 shows the software architecture for the prototype. The prototype performs the following functions:

1. Starts execution of the target program linked with the required libumem.so library. [test.pl, leakmon.pl, target, libumem.so]
2. Creates workload inputs to drive the target program. [workload]
3. Detects memory leaks based on triggering by MSET alarms. [MSET, gcORE | mdb ::findleaks | parse]
4. Frees the memory associated with the detected leaks. [libfree.so]
5. Inserts memory leak faults to aid evaluation of the memory leak detection effectiveness. [libleak.so]

Based on the alarms generated by the MSET module, the leakmon.pl script invokes the gcORE, mdb, and parse components to perform memory leak detection. If leaks are found, the libfree.so component is used to forcibly deallocate the leaks. Periodic checking (without the MSET module) is also possible

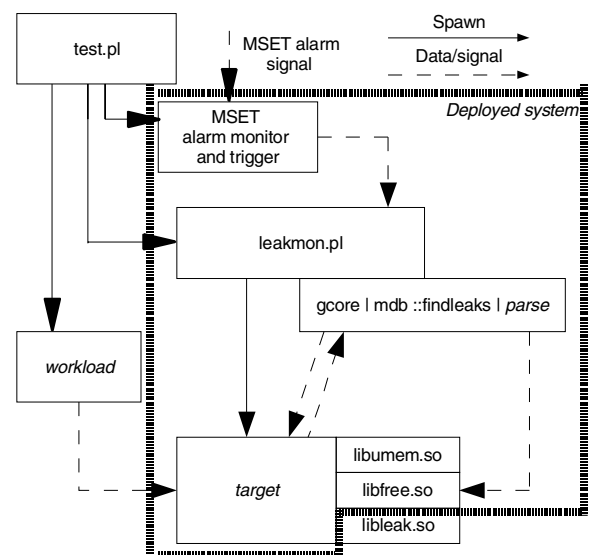


Figure 2: Prototype Software Architecture

in our architecture. In this case, the period between invocations by the `leakmon.pl` script is called the *check period*.

In addition to the deployed system (i.e., the components within the dashed box that would be included in the system when deployed in real usage), the components in Figure 2 include some components needed to implement the experimental testbed, namely `test.pl`, *workload*, and `libleak.so`. The `test.pl` script initiates all activity for the prototype by starting (1) the deployed system and (2) the *workload* script. The *workload* script generates a set of repeatable inputs for the target program, such as creating input files or generating input via standard input or sockets.

Some experiments require injection of memory leak faults in order to provide a known set of memory leaks against which to evaluate the effectiveness of the prototype memory leak detection algorithm. The memory leak injector is implemented in the `libleak.so` shared library. The only fault model currently implemented is an omission to call the `free()` function.

For many operating systems, including Solaris, the order of loading for shared libraries is important for determining the library functions that are dynamically linked to a process. By creating a `free()` function in our `libleak.so` library and loading our library before the `malloc` library where the original `free()` is implemented, we effectively intercept all calls made by the target program to the `free()` function. Under fault-free operation, our `libleak.so` `free()` function simply calls the `malloc` library version. To inject a fault, the `libleak.so` `free()` function returns without calling the `malloc` library version. By injecting faults in this manner, the target program continues with the mistaken belief that the memory has been deallocated. Because the target program never uses the memory nor frees the memory from that point in time onwards, the memory becomes a memory leak.

Furthermore, in order to emulate the real programming bug, in which the call to `free()` has been erroneously omitted from the source code, `libleak.so` remembers the *callsite address* from which `free()` is called. Each callsite is determined to be either a faulty callsite or a fault-free callsite. Thus, once a fault has been injected for a particular callsite, all further calls to `free` from that callsite will be faulty.

The deployed system is initiated via the `leakmon.pl` script, which starts the target program and based on MSET alarms, calls the components that detect memory leaks and correct memory leaks. Stage 2 memory leak detection is performed via the following series of commands:

1. The `gcore` utility temporarily suspends the target program and creates a core image of the entire process memory space. After the core image is obtained, execution of the target program can be resumed.

2. The core image is passed to the `mdb` debugger, which inspects the core image using the `::findleaks` module to detect memory leaks.
3. The raw output from the `mdb` debugger is further parsed by the *parse* script to produce a list of memory leaks, including associated addresses and sizes.

From this list of memory leaks, the `leakmon.pl` script passes the addresses for all memory leaks to the `libfree.so` shared library. The `libfree.so` shared library creates a thread that executes in the context of the target program and is thus able to call the `free()` function and forcibly deallocate the memory leaks. The forcible deallocation of identified memory leaks can be safely performed even though the application has resumed execution because once the application discards all references to a memory object, the application does not know the location of the object and therefore is not capable of regenerating a reference to that object. The `libfree.so` library is implemented in a manner that is similar to the `libleak.so` library. However, to prevent unintended interaction between `libleak.so` and `libfree.so`, which both contain implementations of `free()`, the `libleak.so` library is always loaded first, and both libraries call the version of `free()` in the `malloc` library directly.

3 Experimental Results

One of the main goals for this paper is to show a proof of concept for the proactive run-time memory leak detection and tolerance idea. The prototype described in Section 2 was developed to demonstrate that the idea is effective and to provide a testbed to analyze the effectiveness of the idea as well as the overheads associated with it. This section describes the experiments that were performed on the prototype and the associated results, including demonstration of the effectiveness of our prototype as well as understanding the associated overheads.

3.1 Stage 1 Experiments

We now describe the experimental results from MSET to proactively trigger system memory leak alarms. MSET has been used for proactive annunciation of the incipience or onset of software problems in large, Unix-based multiprocessor servers that are used in business-critical applications as well as in centralized compute resources shared by a large number of users via thin clients. Metrics employed to evaluate the feasibility of MSET for this application include sensitivity, time-to-annunciation, and false-alarm avoidance. Subtle fault injections can be actuated during times periods of large and chaotic multi-user transactional processing to fully evaluate

the sensitivity of MSET for identifying the onset of very subtle anomalies that would ostensibly be masked by dynamic system loads.

Figure 3 shows an example of MSET applied to detect the onset of memory leaks. A data set consisting of 35000 observations each, for 29 software variables (signals), such as memory usage, CPU utilization, paging/swapping activity and disk activity collected from a Solaris-based multiprocessor system approximately was used for the model. These observations were obtained such that daily and weekly variations were covered. From this set, 1000 observations were selected for the training matrix. The training time of the model is linearly proportional to the number of training vectors and to the square of the number of signals in the model [22].

The plot in Figure 3 shows the MSET estimates and observations of “free swap space”. A subtle memory leak was simulated at approximately the midpoint of the observations. The corresponding lower plot shows the SPRT alarms generated by the tool. Sustained positive alarms can be seen soon after the simulated memory leak starts. A proactive memory leak detection and recovery technique such as that described in the following sections is initiated as soon as such alarms occur, preventing a potential catastrophic failure at a later time.

3.2 Stage 2 Application Test Suite

Real applications were used to evaluate the overheads for the prototype in addition to determining the effectiveness of the prototype in detecting injected

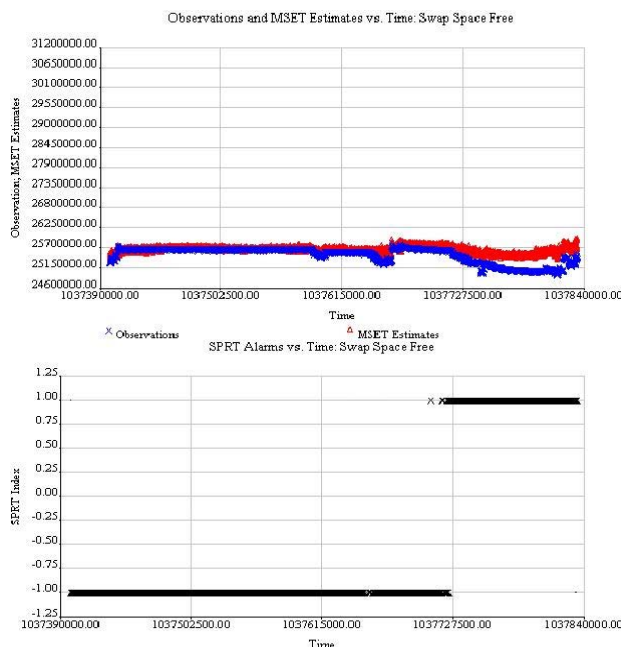


Figure 3: MSET Used to Detect Subtle Memory Leaks

faults. We chose four applications for our application test suite. A summary description of these four applications and their workloads is given in Table 1. The first application is the gzip program (version 1.3.5), which compresses a very large 158MB of ASCII text that represents a Solaris software package. The second application is the GNU tar program (version 1.15.1), which compresses the same 158MB file using bzip2 compression. The third application is the gcc compiler (version 3.4.4), which compiles a single, very large C-language file containing concatenated source code taken from the pine4.58 program. The last application is a Perl script that is interpreted by Perl version 5.8.7 and which contains a series of simple tight loops and subroutine calls.

Table 1: Description of Selected Applications

<i>Application</i>	<i>Description</i>
Gzip	Compress a 158MB file containing ASCII text
Tar	Tar and compress using bzip2 a 158MB file containing ASCII text
Gcc	Compile a selection of large source files from pine4.58
Perl	Execute a Perl script consisting of a series of computer-intensive loops

These four programs were chosen based on the following criteria:

1. The application must allocate and free dynamic memory using the malloc() and free() application programming interface. Otherwise, no memory leaks are possible, and our experiments would not have yielded anything interesting.
2. The execution time of each application must be repeatable. This criterion excluded network-based applications such as web browsers which had execution times that were highly variable. The exact control flow of each application didn't have to be exactly repeatable, but the execution times had to be statistically repeatable for a small number of runs.
3. The execution time for each application must not be too short or too long. To satisfy this requirement, the workload inputs were adjusted until a practical execution time was obtained. Times that are too short prevent the prototype from having enough time to invoke the memory leak detection algorithm, while times that are too long delay experimental results unnecessarily.

Table 2: Single Invocation Overhead Times

<i>Individual Component</i>	<i>Single Invocation Overhead (sec)</i>			
	<i>gzip</i>	<i>tar</i>	<i>gcc</i>	<i>perl</i>
gcore time (sec)	0.03 ± 0.00	0.03 ± 0.00	0.56 ± 0.00	0.05 ± 0.00
core file size (bytes)	1,099,620	678,920	59,247,688	1,869,668
mdb ::findleaks time plus <i>parse</i> (sec)	0.14 ± 0.00	0.10 ± 0.00	15.74 ± 0.02	1.68 ± 0.00
# leaks found	6	6	3292	329

- The application and associated workloads must be capable of being automated. This requirement is necessary to allow repeated invocations in the testbed environment.

When needed, the libleak.so fault injection library described in Section 2 was used. The faulty callsites were chosen randomly. For each application run, up to 10 faulty callsites were chosen. Each callsite was labeled as either fault-free or faulty the first time it was encountered, with a 10% chance of being faulty.

3.2.1 Application Results

Our experiments were performed on a Sun Microsystems SunBlade 1000 with dual 900 MHz UltraSPARC III+ processors and 2 GB of RAM running Solaris 5.9. In the results presented here, only the periodic checking case is discussed. Checking triggered based on MSET alarms (which reduces the overhead) will be presented in a future work.

For the set of four applications in Table 1, we wanted to focus on two types of evaluations: the incurred overhead and the effectiveness of the memory leak detection. Specifically for the evaluation of overheads, we wanted to gain an understanding of the typical overheads that could be expected for our test applications. The first experiments we performed involved running each of the applications in one of three modes:

- Base: The application was executed without the prototype.
- Detect: The application was executed with fault injection, but without any forcible deallocations by libfree.so.
- Free: the application was executed with fault injection and also with forcible deallocations by libfree.so.

The results for these overhead experiments are given in Figure 4. For the Detect and Free modes, the check period was 30 seconds, i.e., the memory leak detection algorithm was invoked every 30 seconds. For each application and mode, the application was executed 10 times. The mean execution times and 95% confidence intervals are shown in Figure 4. For gzip a very small overhead of less than 1% was

measured for the Detect and Free modes. However, gcc and perl showed much larger overheads of approximately 25% and 60%, respectively. To understand why these overheads are so large, we measured the overhead times for single invocations of the gcore and mdb utilities. These results are given in Table 2.

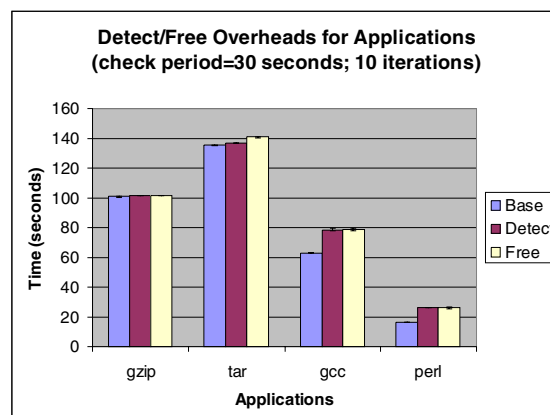


Figure 4: Detect/Free Overheads for Applications

Table 2 shows execution times for the gcore and mdb utilities when executed for each of the four applications. Both gcore and mdb are executed 10 times for each application, and the mean and 95% confidence intervals are given. It should be noted that the execution times for both gcore and mdb depend on the specific moment when they are initiated. For example, gcore saves the target application's memory space to a core file. For larger files, more time is required for gcore to complete. Thus, the numbers presented in Table 2 are intended to provide insight into the overheads seen in Figure 4 rather than to pinpoint the exact execution times for gcore and mdb.

In Table 2, the gcore execution times are given along with the size of the corresponding core file. gcc produces a much larger core file than the other applications. Saving this larger core file to disk takes more time, hence the significantly longer execution time for gcc. However, this extra gcore time for gcc is only half a second greater than for the other applications. The main cause for the overheads seen

in Figure 4 is the execution time for mdb. The mdb execution times along with the corresponding number of memory leaks detected for that particular invocation of mdb are given in the last two rows in Table 2. The time includes both the time directly attributable to mdb as well as the post-processing by the *parse* script. These two times are measured as a single time because they execute simultaneously, with each leak being processed by *parse* immediately when found by mdb. The results show that a larger number of leaks found corresponds to a longer mdb execution time. This is reasonable because each leak requires additional processing by mdb and *parse*. For gcc, a much larger number of leaks must be processed, hence the much larger execution time. Looking back to Figure 4, most of the overheads for all four applications can be accounted for by Table 2.

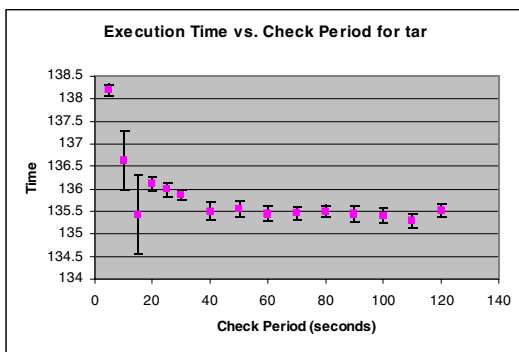


Figure 5: Execution Time vs. Check Period for tar

The 25% and 60% overheads incurred for gcc and perl in Figure 4 are quite high. However, the absolute overhead is incurred each time the memory leak detection is invoked. Thus, we should expect the overhead to be a function of the frequency of memory leak detection and corresponding also a function of the check period. Although the absolute overhead remains the same, the overhead as a percentage of the total execution time decreases as we invoke memory leak detection less often. Figure 5 confirms this intuition by showing the effect of increasing the check period on the total execution time. It should be noted that the range of check periods shown in Figure 5 is not really practical. In practice, the check periods

should probably much greater, perhaps even every few hours. In fact, the check period can even be adaptive. If an application is not showing any memory leaks, the check period can be set to a large period. As the number of leaks detected increases, the period can be adjusted downward. This is the main idea behind the use of a low-overhead Stage 1 to trigger Stage 2 only when a high likelihood of memory leakage exists.

An understanding of the overheads for our prototype is important because the overhead must be manageable for the idea to even be considered for deployment. Still, even with acceptable overhead, the memory leak detection must be effective. For the same four applications used previously, Table 3 shows the effectiveness of the memory leak detection with the mean number of faults injected and detected based on ten runs for each application. Separate numbers are given for running the applications in the Detect and Free modes. There appears to be little noticeable difference between the Detect and Free modes. However, the percentage of injected faults detected varies widely depending on the application.

4 Conclusion

In this paper we have presented an approach to fault tolerance of memory leak faults that leverages an advanced statistical pattern recognition method and existing memory leak detection tools to proactively detect and free memory leaks in deployed systems. Many tools and techniques exist to provide test and debug support in pre-deployment testing labs. These tools have proven to be quite useful and effective in identifying the bugs that lead to memory leaks. Unfortunately, the performance overheads of many of these tools limit their utility in deployment situations.

The contribution of this paper is the presentation of techniques to minimize the performance overhead of garbage collection (i.e., memory leak detection and removal in a run-time system) for a non-cooperative system (i.e., an application or run-time environment that is not engineered to detect or remove unneeded memory). Our technique uses a initial stage based on advanced statistical techniques to restrict execution of the Stage 2 memory leak detection algorithm to

Table 3: Results for Applications with Fault Injection

Application	Detect			Free		
	# Injected	# Detected	% Detected	# Injected	# Detected	% Detected
Gzip	9	8	88%	36	31	87%
Tar	12	9	75%	67	52	78%
Gcc	12037	7371	61%	12140	7453	61%
Perl	4569	361	8%	4469	443	10%

instances where the likelihood of a memory leak are high. The use of Stage 1 helps to relieve some of the demand for the Stage 2 algorithm to have minimal overhead. As part of future work, the possibility of using separate MSET modules for each application is being explored. This will result in checking and debugging applications only when there are alarms from that application, thus reducing the overhead further.

There are also some further possible performance optimizations. First, for simplicity, the core image produced by the gcore utility is saved to disk and then immediately read from disk by the mdb utility. The time to write to disk as well as the disk space can be saved by keeping the core image in memory for use by the mdb utility. However, for large core images, this might introduce some undesirable effects by displacing the target application's virtual pages from physical memory and causing additional swapping from disk. Second, the glue logic to control the mdb utility and post-process the output of the mdb utility is implemented as a Perl script. Additional speedup can be achieved via implementation in a lower-overhead language. Third, the prototype was created by using existing utilities. These utilities were created with other uses in mind, and therefore only a small portion of their full functionality is needed by our prototype. To minimize the overhead even further, custom-designed utilities that combine the minimal gcore and mdb functions needed for the prototype can be created. This would potentially allow greater integration of the two utilities and eliminate some of the inefficiencies due to passing data between two separate processes.

One very interesting use of this memory leak detection and deallocation approach is in high

performance computing systems that employ a large number of processors and memories. Such systems often execute long-running tasks that may suffer the ill effects of memory leaks after much progress has been made. Due to the expense of operating these supercomputers, rerunning the task may incur additional cost. Thus, application of our approach may be useful as long as the overhead is acceptable. One possible source of symbiosis may exist with the checkpointing support often found on supercomputers. As with our approach, checkpointing also requires the temporary suspension of the target application in order to save a consistent snapshot of the memory space. Thus, a single suspension can service both checkpointing and memory leak detection. Also, the same disk file can be used. Because supercomputers often set aside spare processing capacity or include support processors for I/O or other functions, the analysis of the checkpoint file by the memory leak detection algorithm can be done offline, thus minimizing its impact on the target application, which can resume immediately after the checkpoint file is completely saved. There is no difficulty with this delayed invocation of the memory leak detection algorithm because any leaked memory will never be accessed again by the target application by definition and thus can be detected and freed at any point in the future.

5 Acknowledgments

This material is partly based on work supported by the US Defense Advanced Research Project Agency under contract No. NBCH3039002.

Bibliography

- [1] Coverity prevent: Static Source Code Analysis for C and C++, http://www.coverity.com/library/pdf/coverity_prevent.pdf.
- [2] Purify, <http://www-306.ibm.com/software/awdtools/purify/unix/>.
- [3] Parasoft Insure++, <http://www.parasoft.com/jsp/products/home.jsp?product=Insure>.
- [4] Valgrind, <http://valgrind.kde.org>.
- [5] Codework Glowcode, <http://www.codework.com/glowcode/product.html>.
- [6] Compuware BoundsChecker, <http://www.compuware.com/products/devpartner/bounds.htm>.
- [7] LeakTracer, <http://www.andreasen.org/LeakTracer/>.
- [8] Memwatch, <http://www.linkdata.se/sourcecode.html>.
- [9] Brian Willard and Ophir Frieder, "Autonomous Garbage Collection: Resolving Memory Leaks in Long-running Server Applications," *Computer Communications*, vol. 23, 2000, pp. 887-900.
- [10] Hans-Juergen Boehm and Mark Weiser, "Garbage Collection in an Uncooperative Environment," *Software Practice and Experience*, vol. 18, 1988, pp. 807-820.
- [11] Hans-Juergen Boehm, "Space Efficient Conservative Garbage Collection," *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, June 23-25, 1993, pp. 197-206.
- [12] K.C. Gross and K.K. Hoyer, "SIMSPRT: A SAS Code for Simulation of Sequential Probability Tests," *Proceedings of the 18th*

- Annual SAS User's Group International Conference (SUGI-18), May 9-12, 1993.
- [13] K.C. Gross, S. Wegerich, and R.M. Singer, "New Artificial Intelligence Technique Detects Instrument Power Early," *Power Magazine*, 1998, pp. 89-95.
- [14] R. Singer, K.C. Gross, J. Herzog, R. King, and S. Wegerich, "Model-Based Nuclear Power Monitoring and Fault Detection: Theoretical Foundations," *Proceedings of the Intelligent System Application to Power Systems Conference*, July 1997, pp. 60-65.
- [15] K.C. Gross and W. Lu, "Early Detection of Signal and Process Anomalies in Enterprise Computing Systems," *Proceedings of the IEEE International Conference on Machine Learning and Applications (ICMLA)*, June 2002.
- [16] K. Vaidyanathan and K.C. Gross, "Proactive Detection of Software Anomalies through MSET," *Proceedings IEEE Workshop on Predictive Software Models (PSM-2004)*, September 2004.
- [17] A. Wald, *Sequential Analysis*, John Wiley & Sons, 1947.
- [18] Robert Benson, "Identifying Memory Management Bugs Within Applications Using the libumem Library," <http://access1.sun.com/techarticles/libumem.html>, 2003.
- [19] *Solaris Modular Debugger Guide*, Sun Microsystems, Inc., May 2002.
- [20] Jeff Bonwick, "The Slab Allocator: An Object Caching Kernel Memory Allocator," *Proceedings of the Summer 1994 USENIX Conference*, June 6-10, 1994, pp. 87-98.
- [21] Jeff Bonwick and Jonathan Adams, "Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources," *Proceedings of the 2001 USENIX Annual Technical Conference*, June 28-30, 2001, pp. 15-34.
- [22] K. Vaidyanathan and K. C. Gross, "MSET Performance Optimization for Detection of Software Aging," November 2003.