Model Predictive Control for Memory Profiling

Sean Callanan, Radu Grosu, Justin Seyster, Scott A. Smolka, and Erez Zadok *
Stony Brook University

Abstract

We make two contributions in the area of memory profiling. The first is a real-time, memory-profiling toolkit *we call* Memcov *that provides both allocation/deallocation and access profiles of a running program. Memcov requires* no recompilation or relinking *and significantly reduces the barrier to entry for new applications of memory profiling by providing a clean, non-invasive way to perform two major functions: processing of the stream of memory-allocation events in real time and monitoring of regions in order to receive notification the next time they are hit.*

Our second contribution is an adaptive memory profiler and leak detector *called* Memcov$_{MPC}$*. Built on top of Memcov, Memcov$_{MPC}$ uses* Model Predictive Control *to derive an optimal control strategy for leak detection that maximizes the number of areas monitored for leaks, while minimizing the associated runtime overhead. When it observes that an area has not been accessed for a user-definable period of time, it reports it as a potential leak. Our approach requires neither mark-and-sweep leak detection nor static analysis, and reports a superset of the memory leaks actually occurring as the program runs. The set of leaks reported by Memcov$_{MPC}$ can be made to approximate the actual set more closely by lengthening the threshold period.*

1 Introduction

Profiling is a popular technique for finding bugs and inefficiencies in software. Profiling is possible across any interface which has a well-defined set of events. For example, storage profiling can be performed across the file system interface, where the set of events consists of file events such as open, read, and close [10]. Other examples of profiling applications include network profilers, which interpose on an operating system's network APIs to measure the time taken and the amount of data transferred for network protocol interactions [12]—and execution profilers, which sample the CPU program counter to estimate what fraction of a program's runtime is spent in each function [5].

In this paper we present two contributions in this area. The first is a new, general approach to *memory profiling*: a *real-time* profiling toolkit that profiles both allocation/deallocation and access events as the program runs; our toolkit requires *no recompilation or relinking*, and can be used by third-party or custom tools to process the events. The second contribution is an *adaptive* profiler that uses this toolkit to detect memory leaks. Because monitoring an area to see if it is accessed incurs overhead when it is accessed, our profiler monitors commonly used areas less frequently, reducing the overhead from hits on memory areas that are not leaks.

The toolkit we present in this paper, called *Memcov*, significantly reduces the barrier to entry for new applications of memory profiling. Memcov provides a clean, non-invasive way to perform two major functions:

- Reading the stream of memory-allocation events in real time. This allows analysis of total heap consumption, number of allocations per second, distribution of allocation size, and other metrics.
- Monitoring regions in order to receive notifications the next time they are hit. Such a capability supports heap-usage analysis, leak analysis, and working-set profiling.

The profiler, which we call *Memcov$_{MPC}$*, uses the principle of *Model Predictive Control* [4] to derive an optimal strategy for leak detection that maximizes the number of areas monitored for leaks while minimizing the associated runtime overhead. Memcov$_{MPC}$ reports any area for which it has observed no accesses for a user-definable period of time as potential leaks. Our approach requires neither mark-and-sweep leak detection nor static analysis, and we report a superset of the memory leaks actually occurring as the program runs. The set of leaks reported by Memcov$_{MPC}$ can be made to approximate the actual set more closely by lengthening the threshold period.

*Appears in the Proceedings of the NSF Next Generation Software Workshop (NSF NGS 2007) held in conjunction with the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007) 1-4244-0910-1/07/$20.00 ©2007 IEEE.

The rest of the paper is organized as follows. Section 2 discusses Memcov's design. Section 3 describes the Model Predictive Control theory underlying the design of Memcov$_{\mathrm{MPC}}$. Section 4 considers the data Memcov can provide and how Memcov$_{MCP}$ uses it. Section 5 summarizes related work. We conclude in Section 6 and discuss directions for future work.

2 Design

We designed Memcov with three goals in mind:

Flexibility: It should be possible to perform a wide variety of analyses with the tools provided by Memcov. For example, it may be sufficient to test for double-frees or unfreed allocations in the case of simple memory allocations. In another use case, one may want to determine average allocation size. Other use cases may require that LRU statistics be kept for allocations to discover inefficient memory use.

Non-invasiveness: As little profile code as possible should run in the context of the application under investigation. This reduces the possibility of the profiler interfering with application behavior and makes it easier to determine profiling overhead, especially if the profiler is being run on a multicore or multiprocessor computer.

Simplicity: The API for the profiler should be as simple as possible while providing the necessary interfaces to preserve flexibility. We must provide easy access to the stream of memory-allocation, freeing, and access events generated by the application. We must also provide access to all existing memory allocations, allowing the profiler to maintain use-case-specific data about each allocation.

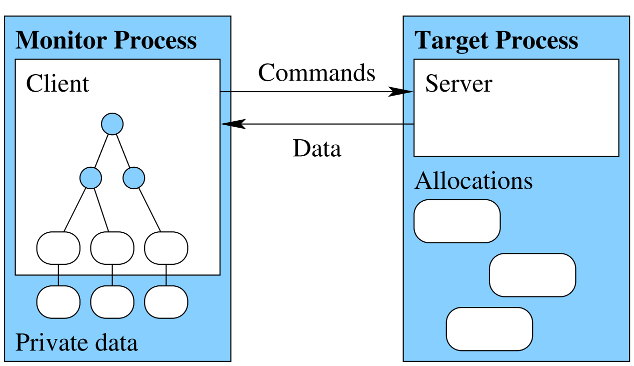

Figure 1. Design of the memory profiler

We illustrate Memcov's high-level architecture in Figure 1. Portions of the memory profiler execute in two contexts: the *target process*, whose memory usage is being profiled, and the *monitor process*, which is doing the profiling. We implemented core functionality for Memcov as a *server* that we embed in the target process by runtime insertion using the loader, and as a *client* that is linked into the monitor process at compile time. This core functionality ensures that `malloc`, `realloc`, `free`, and `calloc` continue to run correctly while keeping track of all existing allocations in a data structure and allowing monitoring of individual memory regions using memory-protection primitives.

The server implements the actual memory allocator back-end in such a way that memory allocations can be monitored individually. The primary challenge here is that the `mprotect` system call, the POSIX interface to the operating system's memory-protection functionality, operates at page granularity; in contrast, most standard memory allocators store multiple small allocations on a single page. This means that if we monitor each small allocation, then all allocations on the same page will also be monitored. We eliminate this granularity mismatch by forcing our allocator back-end to allocate in multiples of a single page. We chose the POSIX `mmap` facility for this purpose; a user-space alternative, `posix_memalign`, exists as well but is very slow.

The client maintains a splay tree containing the base and extent of all existing allocations. This data structure has two consumers: (1) the server, which periodically needs to query extent information for a particular base in order to deallocate, protect, or resize memory regions; and (2) the use-case-specific profiling logic. For the benefit of the profiling logic, the client also maintains one private data pointer per allocation, which can be filled by the profiling logic. The profiling logic can access allocation information by registering a callback that is called each time an event happens, or by obtaining an iterator that we can use to traverse all existing allocations.

The server and the client communicate via a packet-based protocol implemented on top of POSIX pipes. There are two pipes: (1) a *data* pipe on which the server communicates event data and requests extent information; and (2) a *command* pipe on which the client issues commands to the server, instructing it to monitor certain areas and sending the extent information required by the server to implement the memory-allocator back-end.

The protocol is largely asynchronous, as shown in Figure 2. For example, we can allow `free` to return despite the fact that the memory will not be freed until the client has queried its splay tree and determined the extent of the region to be freed. This is because `free` requires eventual deletion, not immediate deletion; behavior on subsequent accesses is undefined [7]. However, one synchronous operation remains: a call to `realloc` requires a round-trip to the

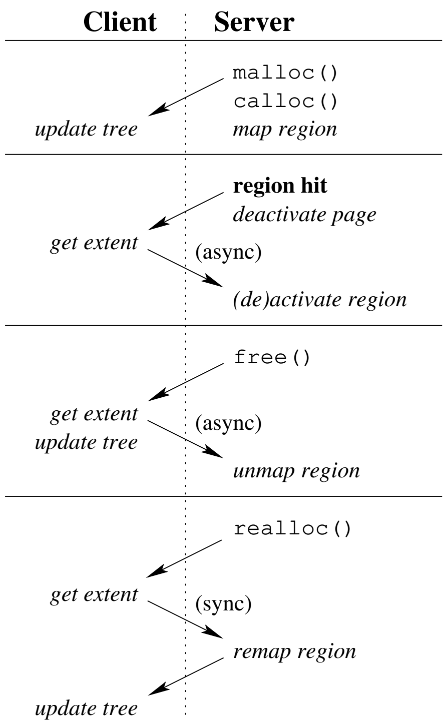

Figure 2. Packets and actions for a selection of memory events

client to obtain the size of the region before resizing. However, in all cases, the Memcov client performs updates to its region database asynchronously. Because of the ordered nature of POSIX pipes, we guarantee that updates are committed before subsequent operations require the updated extent information.

3 Model Predictive Control

Our memory leak detector, Memcov$_{MPC}$, uses Memcov as a backend and implements a Model Predictive Control strategy that seeks to maximize the number of areas monitored for leaks, while minimizing the associated runtime overhead. Upon reaching a given system state s, a model predictive controller [4] performs an online calculation that allows it to explore state trajectories that emanate from s in search of a cost-minimizing control strategy. As explained below, the control action taken by Memcov$_{MPC}$ is to adjust the monitoring rate associated with allocated memory areas.

Strictly speaking, a *memory leak* in an executing program P is a memory allocation that is not accessed by P beyond a certain point in time. In terms of temporal-logic model checking [3], the (negation of the) memory-leak property can be expressed as a liveness property: a temporal-logic formula of the form "Something good eventually happens." Liveness properties, however, are not well-suited for runtime verification, since it is not generally known when the "good" event will eventually occur.

We therefore investigate *bounded memory leaks* with Memcov$_{MPC}$: memory allocations that are not accessed for a time interval T_L. The set of bounded memory leaks is a superset of the set of actual memory leaks, as it may contain allocations that are in fact accessed in the future. Lengthening T_L has the effect of improving the accuracy of the set of leaks reported by Memcov$_{MPC}$. The bounded-memory-leak property corresponds to a bounded liveness property in temporal logic.

Memcov$_{MCC}$ uses the page-monitoring facility provided by the Memcov toolkit to determine when objects are accessed. Areas that are accessed are not leaks, and each memory hit incurs overhead in the form of a page fault and the overhead associated with communicating the event between Memcov's client and server components. Memcov$_{MPC}$ therefore reduces the monitoring rate on areas it observes being accessed, while increasing the monitoring rate on areas it observes to not be accessed. This approach has two advantages. First, it catches more accesses to infrequently used areas, reducing false positives. Second, it enables sorting of areas by how often they are accessed, allowing not only leak detection but also detection of inefficient memory usage.

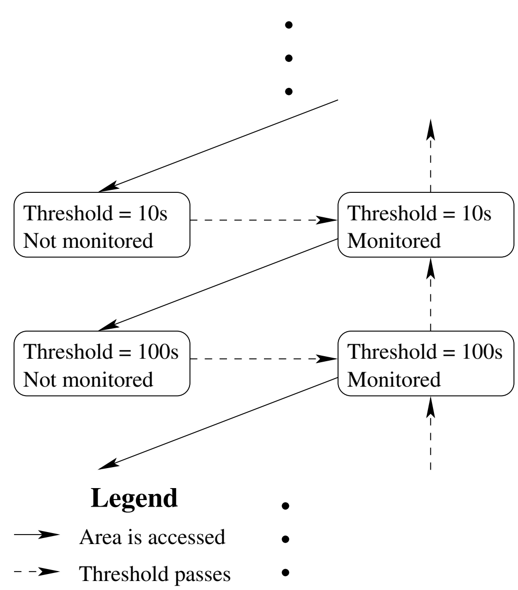

Figure 3. State machine controlling monitoring threshold

Memcov$_{MPC}$ manages the monitoring rate for allocations by adjusting their *monitoring threshold*: the time between an access to an area, which causes deactivation of the area's memory protection, and the reactivation of the protection to detect the next access. The monitoring threshold also determines the amount of time the area is allowed to remain untouched before reducing the monitoring threshold. This leads to a state machine similar to the one shown in Figure 3, differing only in the number of states and their corresponding thresholds.

4 Analysis

In this section, we demonstrate the use of Memcov on the text editor *vim*. First, we show how Memcov can be easily used to build a simple allocation-profiling application. Second, we demonstrate access profiling, first naively and then using Memcov$_{MPC}$, and illustrate the benefit from its adaptive approach to leak detection. Third, we determine the overhead imposed by Memcov's memory-allocator interfaces. In all cases, we ran the benchmarks on a 2-way HyperThreaded 3GHz Pentium 4 with 2GB of RAM, running Linux 2.6.18 and glibc 2.5.

4.1 Using Memcov

We first used the Memcov toolkit to build a profiling application that reports the number of mallocs and frees per unit time. This profiler is a consumer of the stream of event information that the client relays to the profiling logic over the data pipe. We performed no post-processing to obtain the data presented here; the source data for all graphs presented here was emitted as `vim` ran.

We ran a benchmark consisting of a `vim` script that first opened a new file, then repeatedly inserted lines of text at the beginning of the file, wrote the file to disk, deleted every line in the file, again wrote the file to disk, and finally quit. The results of running the profiler on `vim` performing this benchmark are shown in Figure 4; the figure shows the usage statistics for the memory allocator as well as a count of existing allocations. We observe heavy initial memory activity as `vim` starts up. (We discuss the 7-second startup time in Section 4.3.) We then observe a smooth upward trend for memory use, with frees matching but always slightly trailing mallocs, as more and more lines are put into the buffer. Then, the buffer is saved, cleared (causing another spike of activity), and `vim` terminates.

4.2 Leak Analysis and Memcov$_{MPC}$

After analyzing vim's memory allocation behavior, we set out to analyze the use of the allocated memory. We first developed a profiler that uses Memcov to monitor all currently unmonitored areas each second. Because Memcov

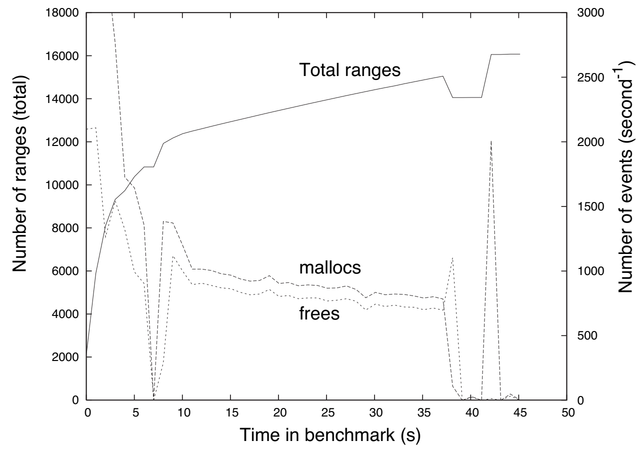

Figure 4. Calls to malloc and free over time for vim.

disables monitoring for areas that are accessed, this means that we are ensuring that each area is hit at most once a second.

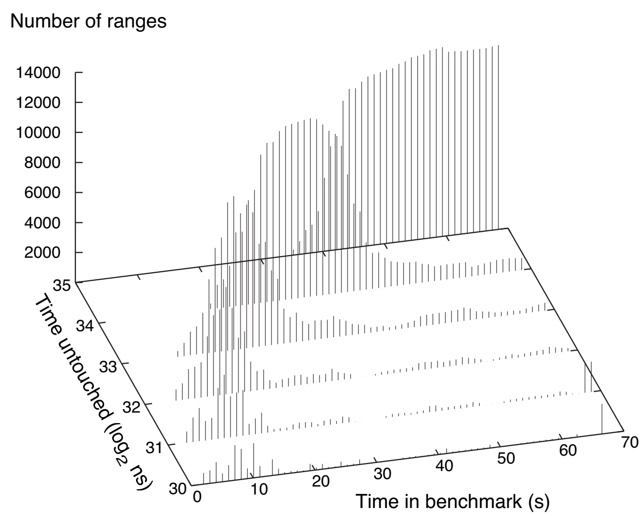

Figure 5. Aging of allocations over time for vim

Figure 5 illustrates the data provided by the profiler. A cross-section along the y-z plane for any point on the x axis is a histogram depicting how many areas have been untouched for less than 1, 2, 4, 8, 16, and 32 seconds at the corresponding time in the benchmark. We observe a gradual aging of memory areas over time, with some initial turnover corresponding to vim start-up but all memory areas gradually aging. By the time vim terminates, 78% of all allocations have been sitting idle for over 16 seconds, and only 0.7% have been accessed within the last 2 seconds.

We then used Memcov$_{MPC}$ to observe how its adaptive approach would affect runtime. Figure 6 shows a histogram over time of the threshold times for all allocations. Initially there is a large group of regions which are accessed frequently enough to cause the threshold to increase from its initial value of 1 second to 10 seconds; as the run continues, however, most of this group is accessed less frequently, and joins the large majority of regions at the shortest interval. This adaptation to usage patterns pays off: the benchmark runs in 71% of the time it takes when monitored naively, even though most regions are monitored with a much higher precision.

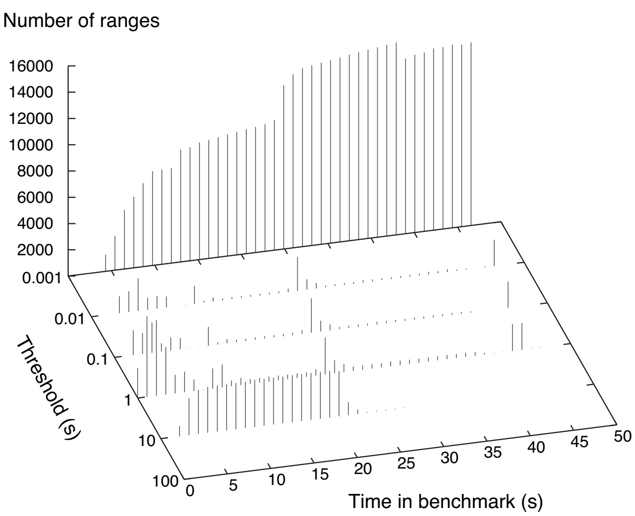

Figure 6. Distributions of threshold times for regions in Memcov$_{\mathrm{MPC}}$

From this data, we observe inefficient use of allocations. Even if they are not actual leaks, the buffers that vim is constantly allocating, as shown in Figure 4, could evidently be re-used since most allocations are sitting unused (as we concluded from Figures 5 and 6).

4.3 Performance

In order to measure the performance overheads incurred by Memcov, we designed a micro-benchmark to test Memcov's memory allocator. The micro-benchmark performs repeated 1-page memory allocations, then touches the allocated memory to defeat a lazy allocator. The micro-benchmark then uses `realloc` to resize each allocation to two pages, touching the second page to make sure it is paged in. Finally, the micro-benchmark frees the memory. We divide by the number of allocations to obtain per-allocation performance metrics for the glibc memory allocator, the Linux `mmap` facility, and Memcov. We benchmarked `mmap` in order to determine how much of Memcov's overhead comes from its allocator as opposed to other factors, such as the asynchronous communication and round-trip for `realloc`. The results are shown in Figure 7.

Operation	glibc	mmap	Memcov
Allocate	13.4	8.1	38.8
Touch	0.27	18.6	24.5
Resize	21.0	131.4	3,008.7
Touch	0.30	19.2	23.3
Free	2.61	26.6	108.3

Figure 7. Allocator performance in thousands of cycles

We observe that Memcov adds large overheads for all memory-allocator operations, most notably realloc (which, as we noted before, requires a round-trip as currently implemented). We also observe that a large portion of this overhead is due to our use of `mmap` as a backend. Although using `mmap` gives us page-aligned memory suitable for individual protection, freeing and resizing areas requires information about the size of the area, which is not provided to glibc. This requires lookups in an auxiliary data structure. Since Memcov does this lookup in another process, it introduces asynchronous behavior: over multiple runs, we observed dramatic fluctuations in the Free phase depending on the interleaving of the dispatch of free notifications to the client and receipt of free commands from the client.

To eliminate this problem, we are working to take `mmap` out of the equation and use the native memory allocator wherever possible. We have implemented the allocator backend for this, and obtained the results shown in Figure 8. We observe dramatic improvements on the resize and free operations, as expected. We now incur an approximately 25-kilocycle (8.3μs) overhead on these operations, which is primarily associated with writing to the data pipe and can be further reduced by using a faster IPC mechanism. We must also alter our page fault handler to support multiple allocations on the same page.

5 Related Work

This paper is primarily intended to present an alternate approach to Chilimbi and Hauswirth's low-overhead temporal profiling tool, SWAT, which they use to perform leak detection [6] by instrumenting memory accesses by particular

Category	glibc	Memcov	Memcov_2
Allocate	13.4	38.8	39.7
Touch	0.27	24.5	0.21
Resize	21.0	3,008.7	60.7
Touch	0.30	23.3	0.31
Free	2.61	108.3	26.3

Figure 8. Effects of swapping out the backend. Memcov_2 is a version of Memcov that uses glibc as a backend.

pieces of code for short periods of time. Using binary translation, SWAT produces instrumented and uninstrumented versions of basic blocks, switching between them to toggle monitoring. It reduces the sampling rate for commonly executed code dynamically. We believe that maintaining sampling rates per allocated area rather than for pieces of code is more appropriate for problems like leaks that are specific to allocations rather than the accessing code; the SWAT approach is perfect, on the other hand, for bounds checking, where the property applies to pieces of code.

The Sun Studio Performance Tools, which began life as the SPARCworks Analyzer, provide visualizations of memory allocation/deallocation activity over the execution time of an application [8, 13]. A *collector* logs memory allocations and deallocations by the program. It also uses *hardware counter overflow profiling* to determine the pages affected by cache misses [9]. The analyzer reads the generated logs and reports as leaks any allocations that are not freed by the time the program terminates.

The `mprof` tool concentrates on determining the cause of heap memory allocations [14]. This consists of a library which is linked into an executable in order to interpose on `malloc` and `free`. Although `mprof` requires the developer to link with a custom library, we presume that the linking could be accomplished using dynamic loader interposing as we do. The library's implementations of the memory allocator functions record the size of the area allocated and the top five addresses from the call chain that led to this allocation, obtained by inspecting the return addresses stored in the stack. This data is post-processed and correlated with symbol information from the binary image of the application under test to produce an *allocation call graph*, in which each function is credited with its callees' allocations.

Solaris's `libumem` provides `malloc` and `free` implementations that collect data about allocated regions [1]. Developers can instrument programs with these functions when the executable is loaded, as Memcov does. The Solaris OS Modular Debugger (MDB) [11] can search for references to allocated areas in a memory dump of a running program. Those areas that are not referenced are flagged as memory leaks. The `libumem` allocation functions can also be configured to pad allocated areas with *red zones* so that MDB can detect bounds violations, and they can similarly fill recently freed areas to detect writes to those areas.

The profiler in IBM's Rational Test RealTime instruments memory allocation and deallocation functions by directly transforming the source code immediately before compilation [2]. The instrumented functions profile allocations using techniques like those used by Solaris's `libumem`. Profiling is not real-time, but instrumented programs can be configured so that events, such as calls to a particular function, trigger the profiler to analyze the running program and report issues, including memory leaks.

6 Conclusions

We have presented and demonstrated Memcov, an innovative toolkit for building memory profilers. It is *flexible*, providing insight not only into an application's memory allocation behavior but also into its memory accesses, thus allowing the detection not only of leaks but of inefficient memory usage. It is *non-invasive*, performing large portions of its logic outside the context of the program being instrumented and requiring no additional instrumentation or analysis phases. Finally, it is *simple*, providing a straightforward and easy-to-understand interface for new profiling applications.

We have also presented Memcov_{MPC}, a leak detector that uses model predictive control to *adapt* to the memory-access patterns of the program under observation. Memcov_{MPC} seeks to reduce the monitoring rate for commonly-accessed regions, thereby allowing it to *zero in* on those areas that are more likely to be leaks. We have demonstrated the performance advantages of Memcov_{MPC} over non-adaptive approaches, showing how it maintains much higher monitoring resolution for most areas while singling out those that would otherwise cause most of the monitoring overhead, and monitoring them less frequently.

7 Acknowledgments

Michael Gorbovitski provided feedback, critique, and support for the architectural model described in Section 2; Scott D. Stoller did the same and also provided valuable comments for this paper.

This work was partially made possible thanks to a Computer Systems Research NSF award (CNS-0509230) and an NSF CAREER award in the Next Generation Software program (EIA-0133589).

References

[1] R. Benson. *Identifying Memory Management Bugs Within Applications Using the libumem Library*, June 2003. `http://access1.sun.com/techarticles/libumem.html`.

[2] J. Campbell. *Memory profiling for C/C++ with IBM Rational Test RealTime and IBM Rational PurifyPlus RealTime*, April 2004. `http://www-128.ibm.com/developerworks/rational/library/4560.html`.

[3] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986.

[4] C. E. Garcia, D. M. Prett, and M. Morari. Model predictive control: Theory and practice—a survey. *Automatica*, 25(3):335–348, 1989.

[5] S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 120–126, June 1982.

[6] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2004)*, pages 156–164, October 2004.

[7] IEEE/ANSI. Information Technology–Test Methods for Measuring Conformance to POSIX–Part 1: System Interfaces. Technical Report STD-2003.1, ISO/IEC, 1992.

[8] M. Itzkowitz. The Sun Studio performance tools. `http://developers.sun.com/sunstudio/articles/perftools.html`, November 2005.

[9] M. Itzkowitz, B. J. N. Wylie, C. Aoki, and N. Kosche. Memory Profiling using Hardware Counters. In *Proceedings of the Supercomputing Conference 2003 (SC2003)*, Phoenix, AZ, USA, November 2003.

[10] J. Ousterhout, H. Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 15–24, Orcas Island, WA, December 1985. ACM.

[11] M. Shapiro. *Solaris Modular Debugger Guide (Solaris 8)*. Fatbrain, October 2000.

[12] T. J. Shepard. TCP packet trace analysis. Technical Report TR-494, Massachusetts Institute of Technology, 1991.

[13] SunSoft. *Performance Tuning an Application*. Sun Microsystems, Inc., August 1994.

[14] B. Zorn and P. Hilfiger. A Memory Allocation Profiler for C and Lisp Programs. In *Proceedings of the Summer 1998 USENIX Conference*, Berkeley, CA, USA, June 1998.