Software Monitoring with Bounded Overhead

Sean Callanan Daniel J. Dean Michael Gorbovitski Radu Grosu Justin Seyster
Scott A. Smolka Scott D. Stoller Erez Zadok
Stony Brook University

Abstract

In this paper, we introduce the new technique of High-Confidence Software Monitoring *(HCSM), which allows one to perform software monitoring with bounded overhead and concomitantly achieve high confidence in the observed error rates. HCSM is formally grounded in the theory of* supervisory control of finite-state automata*: overhead is controlled, while maximizing confidence, by disabling interrupts generated by the events being monitored—and hence avoiding the overhead associated with processing these interrupts—for as short a time as possible under the constraint of a user-supplied target overhead* O_{target}.

HCSM is a general technique for software monitoring in that HCSM-based instrumentation can be attached at any *system interface or API. A generic controller implements the optimal control strategy described above. As a proof of concept, and as a practical framework for software monitoring, we have implemented HCSM-based monitoring for both* bounds checking *and* memory leak detection*. We have further conducted an extensive evaluation of HCSM's performance on several real-world applications, including the Lighttpd Web server, and a number of special-purpose micro-benchmarks. Our results demonstrate how confidence grows in a monotonically increasing fashion with the target overhead, and that tight confidence intervals can be obtained for each target-overhead level.*

1 Introduction

Ensuring the correctness and guaranteeing the performance of complex, long-running software systems, such as operating systems, Web servers and embedded control software, presents unique problems for developers. Errors occur in functions that are called only rarely, and inefficiencies whittle away at performance over the long term. Moreover, it is difficult to replicate all of the environments in which the software may be executed, so many of these problems do not arise until the software is actually deployed.

This state of affairs has led to research in techniques for monitoring software systems during deployment. In this paper, we introduce the new technique of *High-Confidence Software Monitoring* (HCSM). The essential idea is as follows. Given a user-specified *target overhead* O_t, HCSM maximizes the confidence the user has in the monitoring while keeping the actual overhead due to monitoring at levels that never exceed O_t. By maximizing the confidence in the monitoring, we mean that HCSM monitors as many events as possibles within the confines of O_t.

The paper's main contributions can be summarized as follows.

- HCSM is a general technique for software monitoring. HCSM-based instrumentation, which can be attached to *any* system interface or API, maintains an estimate of the rate of accesses to that interface as well as the time spent handling instrumented accesses.
- We have implemented the instrumentation needed for bounds checking as a GCC plug-in, using our newly developed *plug-in architecture for GCC* [5]. For leak detection, we introduce a novel method for detecting stale memory using the virtual memory hardware.
- HCSM is formally grounded in control theory, in particular, the theory of *supervisory control of finite-state automata* [17, 1].

The rest of this paper develops along the following lines. Section 2 explains HCSM's control-theoretic approach to bounding overhead while maximizing confidence. Section 3 presents our architectural framework for HCSM. Section 4 contains our benchmarking results, while Section 5 considers related work. Section 6 offers our concluding remarks and directions for future work.

2 Control-Theoretic Monitoring

Given a process P, henceforth referred to as the *plant*, with controllable input v and output y, and a *reference input* x, the *controller design problem* is that of designing a *controller* Q, with inputs x, y and output v, such that the

composition of Q and P follows the reference input (i.e., y is nearly equal to x) with good dynamic response and small error (see Figure 1).

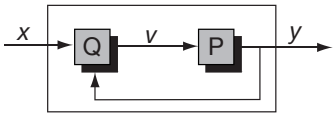

Figure 1. Plant (P) and Controller (Q) architecture.

Runtime monitoring can be beneficially stated as a controller design problem, where the controller is the runtime monitor, the plant is a software application and the reference input x is the *target overhead* o_t. To ensure that the plant is *controllable*, one typically *instruments* the application so that it emits *events* of interest to the monitor. The monitor catches these events, and controls the plant by *enabling* or *disabling* event signaling (interrupts). Hence, the plant can be regarded as a *discrete event process*.

Because of the enabling and disabling of interrupts, the problem we are considering is nonlinear: intuitively, the interrupts signal is multiplied by a control signal that is 1 when interrupts are enabled and 0 otherwise. While a linearization is conceivable, for such nonlinear systems a superior approach is provided in the automata-theoretic setting [17, 1], where the controller design (synthesis) problem is referred to as *supervisory control*.

The main idea of supervisory control we exploit in order to enable and disable interrupts is the synchronization inherent in the *parallel composition* of state machines. In this setting, the plant P is a state machine, the desired outcome (tracking the reference input) is a language L, and the controller design problem is that of designing a controller Q, which is also a state machine, such that the language $L(Q\|P)$ of the composition of Q and P is included in L. In [17, 1] it is shown that this problem is decidable for *finite* state machines.

The monitoring overhead depends on the timing of events and the monitor's per-event processing time. The specification language L therefore consists of *timed words* $a_1, t_1, \ldots, a_n, t_n$ where each a_i is an (access) event and t_i is the time at which a_i has occurred. Consequently, the state machines used to model P and Q must also include a notion of time. In [19], supervisory control is shown to be decidable for *timed automata* [2] and in [18] for *timed transition models*. In our setting, we use a more expressive version of timed automata which allows clocks to be compared to variables, and for such automata decidability is not guaranteed. We therefore design our controller manually, but are currently investigating techniques for the automated synthesis of an "approximate controller." Moreover the controller we are designing consists of the composition of a *global controller* and a set of *local controllers*, one for each plant (object in the application software) that we monitor. Soundness and optimality proofs for these controllers appear in a technical report [4].

Plant model. The plant P (see Figure 2) is described as an extended timed automaton whose alphabet consists of input and output events, and whose locations are labeled with, and whose transition guards may contain, timing constraints of the form $x \sim c$, where x is a *clock*, c is a natural constant, and $\sim$ is one of $<, \leq, =, \geq, >$. We write transition labels in the form `[guard] In / Out, Ass`, where `guard` is a predicate over the automaton's variables, `In` is a sequence of input events of the form `v?e` denoting the receipt of value `e` on channel `x`, `Out` is a sequence of output events of the form `y!a` denoting the sending of value `a` on channel `y`, and `Ass` is a sequence of assignments to the (local) variables. All fields are optional. A transition is *enabled* when its guard is true and the event (if specified) has arrived. A transition is not *forced* to be taken unless letting time flow would violate the condition (invariant) labeling the current location.

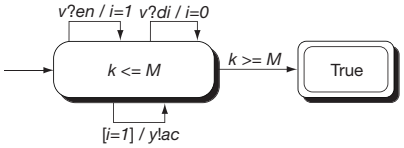

Figure 2. Plant P for a single monitored object.

The plant P has an input channel `v` where it may receive *enable* and *disable* events, denoted `en` and `di`, respectively. It has an output channel `y` where it may send an *access* event `ac`. Upon receipt of `v?di`, the interrupt bit `i` is set to zero preventing the plant from sending further messages. Upon receipt of `v?en`, the interrupt bit is set to one allowing the plant to send messages at arbitrary moments in time. The plant terminates when the maximum monitoring time `M`, a parameter of the model, is reached; i.e., when the clock `k` reaches value `M`. Initially, `i=1` and `k=0`.

Target specification. The specification for a single controlled plant is given as a timed language L. Let `N` denote the natural numbers, $\mathtt{R}^+$ the positive reals, and `A` the set of events. Then:

$$L = \{a_1, t_1, \ldots, a_n, t_n \mid n \in \mathtt{N}, a_i \in \mathtt{A}, t_i \in \mathtt{R}^+\}$$

where the following conditions hold:

1. The average overhead $\overline{o} = (n\, p_a)/(t_n - t_1)$ is $\leq o_t$, where p_a is the average event-processing time.

2. If the strict inequality $\overline{o} < o_t$ holds, then the overhead undershoot is due to time intervals (with low activity) during which all access events are monitored.

The first condition talks only about the mean overhead $\overline{o}$ within a timed word $w \in L$. Hence, various policies for handling overhead, and thus enabling/disabling interrupts, are allowed. The second condition is a *best-effort* condition which guarantees that if the target overhead is not reached, this is only because the plant does not throw enough interrupts. Our policy, described next, satisfies these conditions.

The local controller. For a single monitored plant, our local controller (see Figure 4) disables interrupts by sending event `di` along `v` upon the occurrence of event `ac`, and subsequently enables interrupts by sending event `en` along `v`. Let τ_i be the *time monitoring is on*, i.e., the time between events `en` and `ac`. Let $\mathtt{p_i}$ be the time required to *process* event `ac`, and let $\mathtt{d_i}$ be the *delay time* until monitoring is restarted, i.e., until event `en` is sent again along `v`. We assume that the processing time of an interrupt varies in the interval $[p_m, p_M]$. We refer to the time $\mathtt{c_i} = \tau_\mathtt{i} + \mathtt{p_i} + \mathtt{d_i}$ as a *cycle* and to $o_i = p_i/c_i$ as the *overhead ratio* at `i`. These intervals are shown graphically in Figure 3.

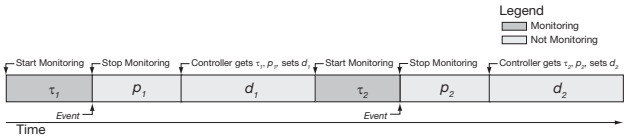

Figure 3. Timeline for a Q ‖ P.

To ensure that $o_i = o_t$ whenever the plant is throwing access events at high rate, the local controller computes $\mathtt{d_i}$ as the least positive real greater than or equal to $\mathtt{p_i}/\mathtt{o_t} - \mathtt{p_i} - \tau_\mathtt{i}$. If the plant throws events at a low rate, then all events are monitored and $\mathtt{d_i} = 0$. Whenever processing of event `ac` is finished, the local controller sends along `u` the processing time `k` to the global controller, which is dicussed below.

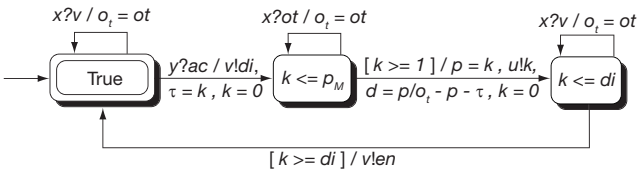

Figure 4. State machine for local controller Q.

The global controller. The local controller Q achieves its target overhead o_t only if the plant P throws events at a sufficiently high rate. Otherwise the mean overhead $\overline{o}$ is less than o_t. In case we monitor a large number of plants P_i simultaneously, it is possible to take advantage of this underutilization of o_t by increasing the overhead o_t of those controllers Q_i associated with plants P_i that throw interrupts at a high rate. In fact, we can scale the target overhead o_t of *all* local controllers Q_i with the same factor λ, as the controllers Q_j of plants P_j with low rate of interrupts will not take advantage of this scaling. We do this every T seconds, a period of time we call the *adjustment-interval*. The periodic adjustment of the local target overheads is the task of the global controller GQ. The architecture of our overall control framework for HCSM is shown in Figure 5.

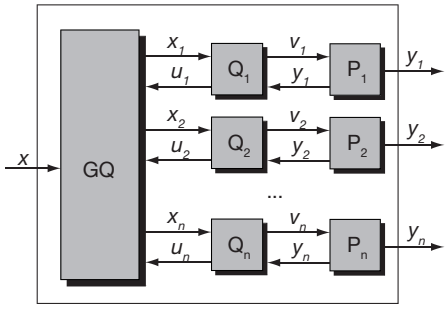

Figure 5. Overall control architecture.

The timed state machine for the global controller `GQ` is given in Figure 6. It inputs on `x` the user-specified target overhead `ot`, which it then assigns to local variable $\mathtt{o_{gt}}$ representing the global target overhead. It outputs ot/n to the local controllers and assigns ot/n to local variable $\mathtt{o_t}$, representing the target overhead for the local controllers. The idea is that the global target overhead is evenly partitioned among the n local controllers. It also maintains an array of total processing time `p`, initially zero, such that $\mathtt{p[i]}$ is the processing time used by local controller $\mathtt{Q_i}$ within the last adjustment-interval of T seconds. Array entry $\mathtt{p[i]}$ is updated whenever $\mathtt{Q_i}$ sends the processing time $\mathtt{p_j}$ of the most recent event a_j; i.e., $\mathtt{p[i]}$ is the sum of the $\mathtt{p_j}$ local controller $\mathtt{Q_i}$ generates during the current adjustment interval.

Whenever the time bound of T seconds is reached, `GQ` computes a scaling factor $\lambda = \sum_{\mathtt{i}=1}^{\mathtt{n}} \mathtt{p[i]}/(\mathtt{T\,n\,o_{gt}})$ as the overall observed processing time divided by the product of T, n and the global target overhead $\mathtt{o_{gt}}$. This factor represents the under- or over-utilization of $\mathtt{o_{gt}}$. The new local target overhead $\mathtt{o_t}$ is then computed by scaling the previous $\mathtt{o_t}$ by λ.

The *target specification language* L_G is defined in a fashion similar to the one for the local controllers, except that the events of the plant `P` are replaced by the events of the parallel composition $\mathtt{P_1} \parallel \mathtt{P_2} \parallel \ldots \parallel \mathtt{P_n}$ of all plants.

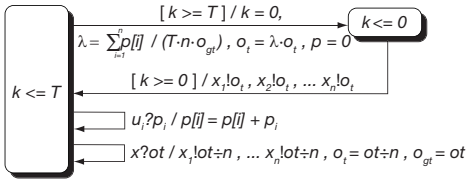

Figure 6. State machine for the global controller.

3 Design

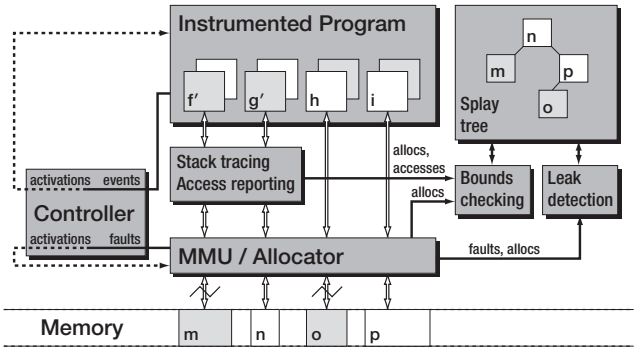

Figure 7. The architecture for our implementation of bounds checking and staleness detection.

In this section, we discuss the two applications that we have implemented for HCSM, namely leak detection and bounds checking. The overall architecture of the system is shown in more detail in Figure 7. The controller regulates delay times for each of these mechanisms, receiving only notifications of hits and incurred overhead. All fault detection and reporting is decoupled from the overhead regulation; both the bounds checker and the leak detector receive information from their respective instrumentation points and consult a shared splay tree of memory areas in performing their functions. We discuss them in more detail in Sections 3.1 and 3.2.

3.1 Memory Under-Utilization

We have implemented an HCSM-based under-utilization detector which identifies areas that are not accessed for a user-definable period of time. We refer to such a time period as a *Non-Accessed Period*, or NAP. Note that we are not detecting areas that are never touched, but rather areas that are not touched for a sufficiently long period of time to raise concerns about memory-usage efficiency.

We introduce a memory-access interposition mechanism called `memcov` that intercepts accesses to particular areas, not accesses by particular instructions. We take advantage of the memory-protection hardware by using the `mprotect` interface, which allows a programmer to control access to a particular memory region. Accesses that violate the access controls set in this way cause segmentation fault signals (`SIGSEGV` on Linux) to be sent to the process in question. By intercepting such faults, which include the faulting address, `memcov` can determine which areas are being accessed by the program and when.

To perform our memory-access interposition, we implement a shared library that replaces memory-allocation functions, such as `malloc` and `free`, with functions that track allocated regions in a splay tree. Each allocation gets its own page, so that `mprotect` can independently control access to it.

3.2 Bounds Checking

The second application for our controller architecture is a more traditional problem: bounds checking. Bounds checking may be broadly defined as ensuring that pointers are dereferenced only when they are *valid*. Our definition of a valid pointer is one that points to a region that

- has been allocated using the system's heap memory allocation functions (notably `malloc`),
- corresponds to some instance of a stack variable (either a local variable or a function parameter), or
- corresponds to a static variable.

We consider any dereferenced pointer to be valid if its target matches the above criteria, regardless of the pointer's type or the region it originally pointed to. This means that we are not required to keep track of each pointer update. Instead, we need only keep track of areas as they are allocated and deallocated. To accomplish this, we use the same splay tree described in Section 3.1. At the entry to each function, we find the stack and static variables used by that function and register them in the splay tree, and at the exit point, we deregister all stack areas used by the function.

To add instrumentation to a program, we use a version of the GNU C compiler that we modified to use plug-ins [5]. Plug-ins are written as normal GCC optimization passes that modify GCC's GIMPLE intermediate representation but can be compiled separately from GCC and loaded dynamically.

Our bounds-checker plug-in, called `meminst`, creates a duplicate copy of every function it instruments. Both copies register and deregister valid memory areas, but the plug-in only adds checks for pointer dereferences in one copy. At

each function call, the controller runs the fully instrumented copy only if bounds checking is enabled for that function.

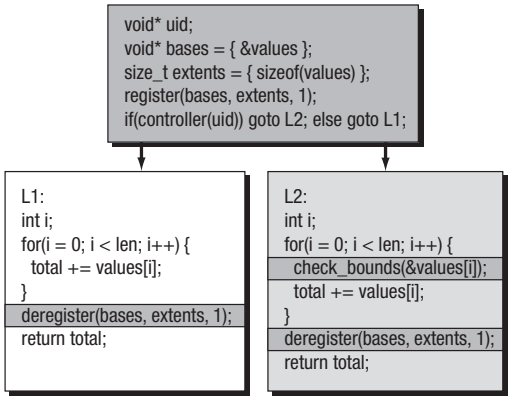

Figure 8. `meminst` adds initial registrations and a call to the controller in a function's first block; the rest is duplicated, and one copy (white) of the function only has deregistrations, whereas the instrumented copy (gray) also includes bounds checking.

Figure 8 shows an abstract representation of a function instrumented by `meminst`.

4 Evaluation

In this section, we describe a series of benchmarks we ran to validate our implementation and determine its runtime characteristics. We will show that HCSM fulfills its goals: not only does it closely adhere to the desired overhead in systems with varying levels of load, but it also observes events at higher rates as it receives more overhead and catches bugs with greater effectiveness. We begin with a real-world demonstration using the Lighttpd Web server [14]. Then we investigate the effectiveness of HCSM by demonstrating its usage with a micro-benchmark that causes bounds violations.

We ran our benchmarks on a group of identically configured machines, each with two 2.8GHz EM64T Intel Xeon processors with 2 megabytes of L2 cache each. The computers each had 1 gigabyte of memory and were installed with the Fedora Core 7 distribution of GNU/Linux. The installed kernel was a vendor version of Linux 2.6.23. We built all packages tested from source: we built the instrumented programs with a custom 4.3-series GCC compiler modified to load plug-ins [5] and other utility programs using a vendor version of GCC 4.1.2. Our Lighttpd benchmarks use Lighttpd version 1.4.18. Graphs that have confidence intervals show the 95% confidence interval over 10 runs, assuming a sample mean distributed according to the Student's-t distribution.

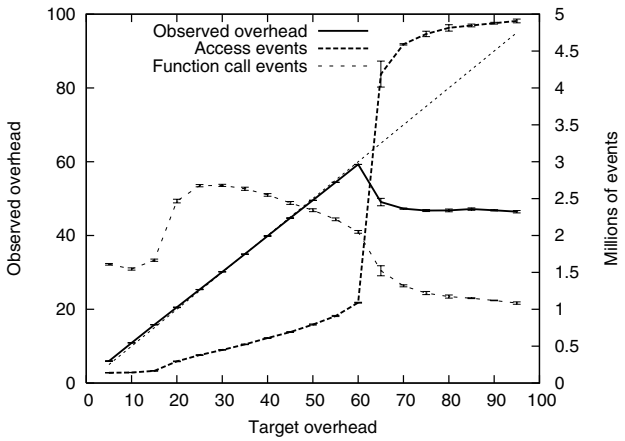

Figure 9. Observed load versus desired load for the Lighttpd server with 25 clients issuing one request per second.

4.1 Lighttpd Benchmark Results

HCSM's main goal is to instrument as much as possible while regulating overhead so that it adheres closely to the user's specification. Since our theoretical result in Section 2 shows this to be achievable, any deviation of the measured performance results from user specification must arise from implementation limitations.

The most obvious limitation is saturation: regardless of the kind of instrumentation, once the instrumented system hits a bottleneck (usually the processor), it will reach a peak service level. More instrumentation will simply reduce the service level, and as a result we will not be able to observe more than a particular threshold of events.

Figure 9 shows observed overhead versus target overhead for our load benchmark of the Lighttpd server, with both bounds checking and memory-under-utilization detection turned on. We ran Lighttpd with target overheads from 5% to 100% in increments of 5%. The solid line shows the observed percent overhead (y axis), which should ideally adhere to the $y = x$ line. The dotted and dashed lines show the number of processed function call events and memory access events respectively, in millions of events (y2 axis).

We use the `curl-loader` tool to hit the server with one request per second from 25 simulated clients. We observe roughly linear growth for target overhead settings between 5% and 60%, with the saturation point lying at 60%.

We observe that saturation accompanies a rapid increase in the number of memory access events observed. At a point

between 60% and 70% target overhead, we observed that the CPU becomes saturated: we measured CPU usage to be at its maximum. Fluctuations at higher target overheads are due to the controller attempting to get more overhead out of a CPU that is pegged at 100% usage. With a saturated CPU, it is not possible to increase observed overhead by increasing the monitoring rate for functions. The global controller consequently increases the adjusted target overhead dramatically, and the individual memory area controllers instrument more aggressively to match the higher target. At higher targets we enter a mode that is almost streaming: all but 3–4 areas are monitored at a time, which actually reduces overhead per observed access by decreasing contention on an internal queue that the controller uses to keep track of the subjects that are waiting to be activated. However, the CPU is still saturated, so the cost of processing this overhead is borne by reducing per-function instrumentation or by sacrificing observed overhead.

During our benchmarking, we did not observe any bounds violations resulting from bugs in Lighttpd. However, we observed a number of NAPs (non-accessed periods, see Section 3). A comparatively large amount of memory, about 180 kilobytes, goes unused for almost all of the program's run. Lighttpd is intended as an embedded Web server with a low memory footprint—its total heap footprint is 540 kilobytes—so the unused 180 kilobytes are of particular interest and comprise a significant reduction in Lighttpd's heap memory footprint (one-third less). We have verified that at least some of these areas come from a preloaded MIME type database that could be loaded incrementally on demand.

4.2 Micro-Benchmark Results

Having seen the effectiveness of HCSM at catching memory leaks, we turn to the effectiveness of HCSM at catching bounds violations. We designed a micro-benchmark that runs for ten seconds, accessing a single memory area as fast as it can. Ten times a second, it issues an out-of-bounds access. This micro-benchmark allows us to examine the performance of HCSM in more detail.

Figure 10 shows our effectiveness at detecting bounds violations in this micro-benchmark for different target overhead settings. The solid line shows the percent of bounds violations caught, and the dotted line shows how many events HCSM observed overall. Initially, we observe a linearly increasing number of accesses, which saturates near 100% of accesses observed, confirming that we are not only achieving our overhead targets, we are in fact getting something for that overhead—we are achieving the goal explained in Section 2: to monitor as much as possible given the overhead constraints.

In each of our tests, the number of bounds violations

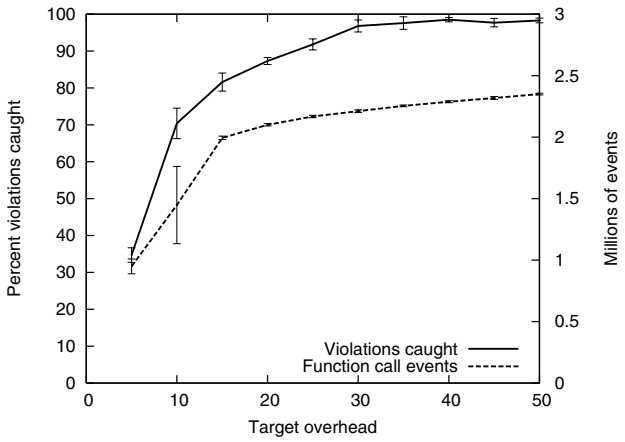

Figure 10. Number of function executions bounds-checked and number of bounds violations caught, versus target overhead for the micro-benchmark.

observed increases evenly throughout, consistent with the micro-benchmark's highly regular behavior. This is in contrast to adaptive sampling tools [12] that reduce overhead over time, making violations occurring later less likely to be caught than early violations.

5 Related Work

In this paper, we have presented techniques that touch on a variety of fields of study. In this section, we will survey some of the other approaches that exist in these fields, and describe what distinguishes our contribution from them. We will discuss the fields of control theory, bounds checking, and leak detection, starting with two systems that are closely related to HCSM.

Chilimbi and Hauswirth [12] have proposed a sampling-based technique for detecting memory leaks that approximates memory-access sampling through code sampling at the level of basic blocks. Their tool, SWAT, records memory accesses in each monitored basic block. Low overhead is ensured by reducing the sampling rate of blocks that generate a high rate of memory accesses. The aim of their sampling policy is, however, to reduce overhead over time; once monitoring is reduced for a particular piece of code, it is never increased again, regardless of that code's future memory-access behavior. Also, because they approximate sampling of memory accesses by sampling basic-block executions, and because a leak is most commonly associated with a memory allocation and not with a basic block, there is no clear association between the sampling rate for blocks and the confidence in any particular leak.

Artemis [8] also reduces overhead from runtime checks by enabling them for only some function executions. In order to observe as many behaviours as possible, Artemis always runs a function with instrumentation when the function runs in a *context* that it has not seen before, where a function execution's context consists of the values of global variables and arguments. The user cannot provide a bound on overhead, however, so Artemis may still instrument too aggressively when functions run in many different contexts.

Control theory. The classic theory of digital control for linear systems is discussed extensively in [9]. The theoretical treatment of the automatic synthesis of a maximal controller within supervisory control is given in [17, 1]. The same is discussed within a timed framework in [17, 1]. In this paper we go beyond the theoretical limitations of automated controller synthesis for timed automata and timed transition models in [17, 1], by manually constructing the controller and proving its soundness and optimality. Note that extracting an optimal controller from the maximal controller is generally a hard problem. Moreover, we apply the supervisory control theory to two nontrivial applications: leak detection and bounds checking. To the best of our knowledge, this has not been done before.

Bounds checking. Keeping track of valid areas associated with particular pointers is usually done with the help of the compiler [7]. One approach is to keep information about valid areas in out-of-band data structures [11], which can be checked at each memory access, which has the advantage that it can tolerate external code accessing the pointers.

The alternative is performing bounds checking at the memory, rather than at the pointer. One technique is to place canary values around critical areas and verify them before they are used [6]. Alternatively, areas can be surrounded with pages that have been monitored using the system's virtual memory hardware [15], which is most effective for heap areas because it is low-overhead but requires space on each side of the allocation for a full page of untouchable memory. Finally, a recent technique is to randomize the allocator [3]. If multiple copies of a program are executed with different memory layouts, deviations in their behavior will indicate pointer manipulations that took pointers out of their designated zones.

Leak detection. Though most general leak detection techniques are dynamic, overhead can be significantly reduced using static analysis [13]. Purely static approaches do exist [10], but must deal with the problem of false positives. Our approach allows us to measure the use of even those allocations that are properly deallocated. This sort of profiling is nearly impossible to perform statically.

Another system that uses hardware support but achieves finer-grained allocation size is SafeMem [16], which repurposes ECC memory's built in error-checking to perform access detection.

6 Conclusions and Future Work

We have presented High Confidence Software Monitoring (HCSM), an approach to overhead control for the runtime monitoring of instrumented software. HCSM is high-confidence because it monitors as many events as possible without exceeding a target overhead. The key to HCSM's performance is an underlying control strategy based on an optimal controller for a nonlinear control problem represented in terms of the composition of timed automata.

Using HCSM as a foundation, we have developed two sophisticated monitoring tools: a memory staleness detector and a bounds checker. Both the per-area and per-function checks in these detectors are enabled and disabled by the same generic controller, which achieves a desired target overhead with both of these systems running.

Our benchmarking results demonstrate that it is possible to perform correctness monitoring of large software systems with fixed overhead guarantees. We also demonstrated the effectiveness of our system at detecting real-world issues: we demonstrated that one-third of the Lighttpd Web server's heap footprint is unused.

Future work. Because HCSM is such a general approach, there are many potential uses for it. Such varied techniques as integer value profiling, lockset profiling and checking, runtime type checking, and intrusion detection would benefit from being coupled with controllable overhead regulation. Some applications could also manage disk or network time instead: a background file-system consistency checker could use the HCSM controller to ensure that it gets only a specific fraction of disk time.

Kernel services that run asynchronously also benefit from a fixed-overhead approach. For example, a heavyweight packet filter may be willing to invoke constant overhead on packets coming in to prevent Denial-of-Service attacks, but ensure that its operation does not itself cause a Denial of Service under heavy load.

Apart from specific applications such as the ones mentioned above, we also believe that several general improvements should be made to HCSM. It should be extended to handle events that must be processed for the rest of the instrumentation to work; for example, a runtime type checker must add some overhead to allocation functions to keep track of types for heap areas, so the type-checking operations should be regulated keeping the overhead from the allocations in mind.

7 Acknowledgments

This work was partially made possible thanks to a Computer Systems Research NSF award (CNS-0509230) and NSF CAREER awards in the Next Generation Software program (CNS-0113589), and the Embedded and Hybrid Systems program (CCR-0133583). We also thank Rob Johnson, Annie Liu, and Paul Talamo from Stony Brook University for their advice and feedback, and Eric Christopher and Diego Novillo from the GCC project for assistance porting our GCC plug-ins to mainline GCC.

References

[1] A. Aziz, F. Balarin, R. K. Brayton, M. D. Dibenedetto, A. Sladanha, and A. L. Sangiovanni- Vincentelli. Supervisory control of finite state machines. In P. Wolper, editor, *7th International Conference On Computer Aided Verification*, volume 939, pages 279–292, Liege, Belgium, 1995. Springer Verlag.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] E. D. Berger and B. G. Zorn. DieHard: probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*, Ottawa, Canada, June 2006.

[4] S. Callanan, D. J. Dean, M. Gorbovitski, R. Grosu, J. Seyster, S. A. Smolka, S. D. Stoller, and E. Zadok. High-confidence software monitoring with bounded overhead. Technical Report FSL-07-03, Computer Science Department, Stony Brook University, November 2007. `www.fsl.cs.sunysb.edu/docs/memcov-tr/memcov.pdf`.

[5] S. Callanan, D. J. Dean, and E. Zadok. Extending GCC with Modular GIMPLE Optimizations. In *Proceedings of the 2007 GCC Developers' Summit*, Ottawa, Canada, July 2007.

[6] C. Cowan, C. Pu, D. Maier, H. Hintongif, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the Seventh USENIX Security Symposium*, pages 63–78, San Antonio, TX, January 1998.

[7] F. C. Eigler. Mudflap: Pointer Use Checking for C/C++. In *Proceedings of the First Annual GCC Developers' Summit*, pages 57–70, Ottawa, Canada, May 2003.

[8] L. Fei and S. P. Midkiff. Artemis: practical runtime monitoring of applications for execution anomalies. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*, Ottawa, Canada, June 2006.

[9] G. Franklin, J. Powell, and M. Workman. *Digital Control of Dynamic Systems, Third Edition*. Addison Wesley Longman, Inc., 1998.

[10] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A System and Language for Building System-Specific, Static Analyses. In *ACM Conference on Programming Language Design and Implementation*, pages 69–82, Berlin, Germany, June 2002.

[11] R. Hastings and B. Joyce. Purify: fast detection of memory leaks and access errors. In *Proceedings of the Winter 1992 USENIX Conference*, pages 125–138, San Francisco, CA, January 1992.

[12] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. *SIGARCH Comput. Archit. News*, 32(5):156–164, 2004.

[13] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '03)*, San Diego, CA, June 2003.

[14] J. Kneschke. Lighttpd. `http://www.lighttpd.net/`, 2003.

[15] B. Perens. *efence(3)*, April 1993. `linux.die.net/man/3/efence`.

[16] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA 2005)*, San Francisco, CA, February 2005.

[17] P. Ramadge and W. Wonham. Supervisory control of a class of discrete event systems. *SIAM J. Control and Optimization*, 25(1):206–230, 1987.

[18] P. Ramadge and W. Wonham. Supervisory control of timed discrete-event systems. *IEEE Transactions on Automatic Control*, 38(2):329–342, 1994.

[19] H. Wong-Toi and G. Hoffmann. The control of dense real-time discrete event systems. In *Proc. of 30th Conf. Decision and Control*, pages 1527–1528, Brighton, UK, 1991.