Int J Softw Tools Technol Transfer (2012) 14:327–347
DOI 10.1007/s10009-010-0184-4

Software monitoring with controllable overhead

Xiaowan Huang · Justin Seyster · Sean Callanan · Ketan Dixit · Radu Grosu · Scott A. Smolka · Scott D. Stoller · Erez Zadok

Published online: 29 December 2010

Abstract We introduce the technique of software monitoring with controllable overhead (SMCO), which is based on a novel combination of supervisory control theory of discrete event systems and PID-control theory of discrete time systems. SMCO controls monitoring overhead by temporarily disabling monitoring of selected events for as short a time as possible under the constraint of a user-supplied target overhead o_t. This strategy is optimal in the sense that it allows SMCO to monitor as many events as possible, within the confines of o_t. SMCO is a general monitoring technique that can be applied to any system interface or API. We have applied SMCO to a variety of monitoring problems, including two highlighted in this paper: integer range analysis, which determines upper and lower bounds on integer variable values; and non-accessed period detection, which detects stale or underutilized memory allocations. We benchmarked SMCO extensively, using both CPU- and I/O-intensive workloads, which often exhibited highly bursty behavior. We demonstrate that SMCO successfully controls overhead across a wide range of target overhead levels; its accuracy monotonically increases with the target overhead; and it can be configured to distribute monitoring overhead fairly across multiple instrumentation points.

Keywords Software instrumentation · Supervisory control

1 Introduction

Ensuring the correctness and guaranteeing the performance of complex software systems, ranging from operating systems and Web servers to embedded control software, presents unique problems for developers. Errors occur in rarely called functions and inefficiencies hurt performance over the long term. Moreover, it is difficult to replicate all of the environments in which the software may be executed, so many such problems arise only after the software is deployed. Consequently, testing and debugging tools, which tend to operate strictly within the developer's environment, are unable to detect and diagnose all undesirable behaviors that the system may eventually exhibit. Model checkers and other verification tools can test a wider range of behaviors but require difficult to develop models of execution environments and are limited in the complexity of programs they can check.

This situation has led to research in techniques to monitor deployed and under-development software systems. Such techniques include the DTrace [6] and DProbes [14] dynamic tracing facilities for Solaris and Linux, respectively. These frameworks allow users to build debugging and profiling tools that insert *probes* into a production system to obtain information about the system's state at certain execution points.

DTrace-like techniques have two limitations. First, they are *always on*. Hence, frequently occurring events can cause significant monitoring overhead. This raises the fundamental question: *is it possible to control the overhead due to software monitoring while achieving high accuracy in the monitoring results?* Second, these tools are *code oriented*: they interpose on execution of specified instructions in the code; events such as accesses to specific memory regions are difficult to track with these tools, because of their interrupt-driven nature. Tools such as Valgrind allow one to instrument memory accesses, but benchmarks have shown a fourfold increase in runtimes even without any instrumentation [17]. This raises a related question: *is it possible to control the*

X. Huang · J. Seyster (✉) · S. Callanan · K. Dixit · R. Grosu ·
S. A. Smolka · S. D. Stoller · E. Zadok
Stony Brook University, Stony Brook, USA
e-mail: jseyster@cs.stonybrook.edu

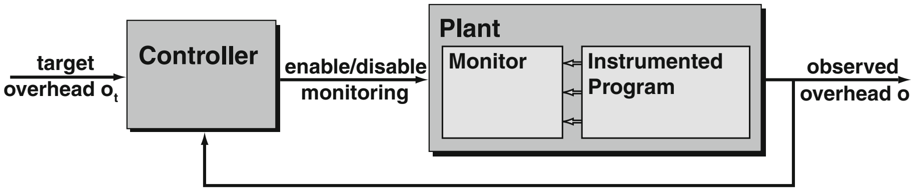


Fig. 1 Structure of a typical SMCO application

overhead due to monitoring accesses to specific memory regions?

To answer the first question, we introduce the new technique of software monitoring with controllable overhead (SMCO). To answer the second question, we instrument the application program, such that SMCO can exploit the *virtual memory hardware*, to selectively monitor memory accesses.

As the name suggests, SMCO is formally grounded in control theory, in particular, a novel combination of supervisory control of discrete event systems [1,15] and proportional-integral-derivative (PID) control of discrete time systems [18]. Overhead control is realized by temporarily disabling interrupts generated by monitored events, thus avoiding the overhead from processing these interrupts. Moreover, such interrupts are disabled for as short a time as possible so that the number of events monitored, under the constraint of a user-supplied *target overhead* o_{t}, is maximized.

Our main motivation for using control theory is to avoid an ad hoc approach to overhead control, in favor of a much more rigorous one based on the extensive available literature [9]. We also hope to show that significant benefits can be gained by bringing control theory to bear on software systems and, in particular, runtime monitoring of software systems.

The structure of a typical SMCO application is illustrated in Fig. 1. The main idea is that given the target overhead o_{t}, an SMCO controller periodically sends enable/disable monitoring commands to an instrumented program and its associated monitor in such a way that the monitoring overhead never exceeds o_{t}. Note that the SMCO controller is a *feedback controller*, as the current observed overhead level is continually fed back to it. This allows the controller to carefully monitor the observed overhead and, in turn, disable monitoring when the overhead is close to o_{t} and, conversely, enable monitoring when the likelihood of the observed overhead exceeding o_{t} is small. Also, note that the instrumented program sends events of interest to the monitor as they occur, e.g., memory accesses and assignments to variables.

More formally, SMCO can be viewed as the problem of designing an optimal controller for a class of nonlinear systems that can be modeled as the parallel composition of a set of extended timed automata (see Sect. 2). Furthermore, we are interested in proportional-integral-derivative (PID) controllers. We consider two fundamental types of PID controllers: (1) a single, integral-like *global controller* for all monitored objects in the application software; and (2) a *cascade controller*, consisting of an integral-like *primary controller* which is composed of a number of proportional-like *secondary controllers*, one for each monitored object in the application. The main function of the primary controller is to control the set point of the secondary controllers.

We use the suffix "-like" because our PID controllers are event driven, as in the setting of discrete-event supervisory control, and not time-driven, as in the traditional PID control of continuous or discrete time systems. The primary controller and the global controller are integral-like because the control action is based on the sum of recent errors (i.e., differences between target and observed values). The secondary controllers are proportional-like because the controller's output is directly proportional to the error signal.

We use both types of controllers because of the following trade-offs between them. The global controller features relatively simple control logic and hence is very efficient. It may, however, undersample infrequent events. The cascade controller, on the other hand, is designed to provide fair monitoring coverage of all events, regardless of their frequency.

SMCO is a general runtime-monitoring technique that can be applied to *any* system interface or API. To substantiate this claim, we have applied SMCO to a number of monitoring problems, including two highlighted in this paper: *integer range analysis*, which determines upper and lower bounds on the values of integer variables; and non-accessed period (NAP) detection, which detects stale or underutilized memory allocations. Integer range analysis is code-oriented, because it instruments instructions that update integer variables, whereas NAP detection is memory oriented, because it intercepts accesses to specific memory regions.

The source-code instrumentation used in our integer range analysis is facilitated by a technique we recently developed called compiler-assisted instrumentation (CAI). CAI is based on a *plug-in architecture for GCC* [5]. Using CAI, instrumentation plug-ins can be separately compiled as shared objects, which are then dynamically loaded into GCC. Plug-ins have read/write access to various GCC internals, including abstract syntax trees (ASTs), control flow graphs (CFGs) and static single-assignment (SSA) representations.

The instrumentation in our NAP detector makes novel use of virtual memory hardware (MMU) by using the `mprotect` system call to guard each memory area suspected of being under-utilized. When a protected area is accessed, the MMU generates a segmentation fault, informing the monitor that the area is being used. If a protected area remains unaccessed for a period of time longer than a user-specified threshold (the NAP length), then the area is considered stale. SMCO controls the total overhead of NAP detection by enabling and disabling the monitoring of each memory area appropriately.

To demonstrate SMCO's ability to control overhead while retaining accuracy in the monitoring results, we performed a variety of benchmarking experiments involving real applications. Our experimental evaluation considers virtually all aspects of SMCO's design space: cascade versus global controller, code-oriented versus memory-oriented instrumentation, CPU-intensive versus I/O-intensive workloads (some of which were highly bursty). Our results demonstrate that SMCO successfully controls overhead across a wide range of target overhead levels; its accuracy monotonically increases with the target overhead and it can be configured, using the cascade controller, to fairly distribute monitoring overhead across multiple instrumentation points. Additionally, SMCO's *base overhead*, the overhead observed at $o_{\mathrm{t}} = 0$, is a mere 1–4%.

The rest of the paper is organized as follows. Section 2 explains SMCO's control-theoretic foundations. Section 3 describes our architectural framework and the applications we developed. Section 4 presents our benchmarking results. Section 5 discusses related work. We conclude in Sect. 6 and discuss future work.

2 Control-theoretic monitoring

The *controller design problem* is the problem of devising a controller Q that regulates the input v to a process P (henceforth referred to as the *plant*) in such a way that P's output y adheres to a *reference input* x with good dynamic response and small error; see the architecture shown in Fig. 2.

Runtime monitoring with controllable overhead can beneficially be stated as a controller design problem. The controller is a feedback controller that observes the monitoring overhead; the plant comprises the runtime monitor and the application software, and the reference input x to the controller is given by the user-specified *target overhead* o_t. This structure is depicted in Fig. 1. To ensure that the plant is *controllable*, one typically *instruments* the application and the monitor so that they emit *events* of interest to the controller. The controller catches these events and controls the plant by *enabling* or *disabling* monitoring and event signaling. Hence, the plant can be regarded as a *discrete event process*.

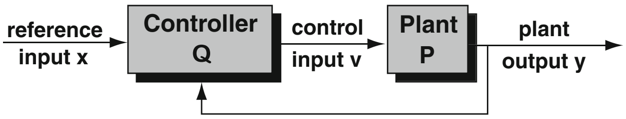


Fig. 2 Plant (P) and controller (Q) architecture

In runtime monitoring, overhead is the measure of how much longer a program takes to execute because of monitoring. If an unmodified and unmonitored program executes in time R and executes in total time $R + M$ with monitoring, we say that the monitoring has overhead M/R.

Instead of controlling overhead directly, it is more convenient to write the SMCO control laws in terms of *monitoring percentage*: the percentage of execution time spent monitoring events, which is equal to $M/(R + M)$. Monitoring percentage m is related to the traditional definition of overhead o by the equation $m = o\,/\,(1 + o)$. The user-specified target monitoring percentage (UTMP) m_{t} is derived from o_{t} in a similar manner; i.e., $m_{\mathrm{t}} = o_{\mathrm{t}}\,/\,(1 + o_{\mathrm{t}})$.

The classical theory of digital control [9] assumes that the plant and the controller are linear systems. This assumption allows one to semi-automatically design the controller by applying a rich set of design and optimization techniques, such as the Z-transform, fast Fourier transform, root-locus analysis, frequency response analysis, proportional-integrative-derivative (PID) control, and state–space optimal design. For nonlinear systems, however, these techniques are not directly applicable, and various linearization and adaptation techniques must be applied as pre- and post-processing, respectively.

The problem we are considering is nonlinear, because of the enabling and disabling of interrupts. Intuitively, the interrupt signal is multiplied by a control signal, which is 1 when interrupts are enabled and 0 otherwise. Although linearization is one possible approach for this kind of nonlinear system, automata theory suggests a better approach, recasting the controller design (synthesis) problem as one of *supervisory controller design* [1,15].

The main idea of supervisory control we exploit 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. This problem is decidable for *finite* state machines [1,15].

Monitoring percentage depends on the timing (frequency) of events and the monitor's per-event processing time. The specification language L therefore consists of *timed words* $a_1, t_1, \ldots, a_l, t_l$ where each a_i is an (access) event that occurs at time t_i. Consequently, the state machines used to model P and Q must also include a notion of time. Previous work has shown that supervisory control is decidable for *timed automata* [2,19] and for *timed transition models* [16].

Modeling overhead control requires, however, the use of more expressive, extended timed automata (see Sect. 2.2), and for such automata decidability is lost. The lack of decidability means that a controller cannot be automatically synthesized. This, however, does not diminish the usefulness of control theory. On the contrary, this theory becomes an indispensable guide in the design of a controller that satisfies a set of constraints. In particular, we use control theory to develop a novel combination of supervisory and PID control. As in classical PID control, the error from a given set point (and the integral and derivative of the error) is employed to control the plant. In contrast to classical PID control, the computation of the error and its associated control happens in our framework on an event basis, instead of a fixed, time-step basis.

To develop this approach, we must reconcile the seemingly incompatible worlds of event- and time-based systems. In the time-based world of discrete time-invariant systems, the input and the output signals are assumed to be known and available at every multiple of a fixed sampling interval Δt. Proportional control (P) continually sets the current control input $v(n)$ as proportional to the current error $e(n)$ according to the equation $v(n) = ke(n)$, where n stands for $n\Delta t$ and $e(n) = y(n) - x(n)$ (recall that v, x, and y are depicted in Fig. 2). Integrative control (I) sums the previous and current error and sets the control input to $v(n) = k \sum_{i=0}^{n} e(n)$.

In contrast, in the event-based world, time information is usually abstracted away, and the relation to the time-based world, where controller design is typically done, is lost. However, in our setting the automata are timed, that is, they contain clocks, ticking at a fixed clock interval Δt. Thus, events can be assumed to occur at multiples of Δt, too. Of course, communication is event based, but all the necessary information to compute the proper control value $v(t)$ is available, whenever an event is thrown at a given time t by the plant.

We present two controller designs with different tradeoffs and correspondingly different architectures. Our *global controller* is a single controller responsible for all objects of interest in the monitored software; for example, these objects may be functions or memory allocations, depending on the type of monitoring being performed. The global controller features relatively simple control logic and hence is very efficient: its calculations add little to the observed overhead. It does not, however, attempt to be fair in terms of monitoring infrequently occurring events. Our *cascade controller*, in contrast, is designed with fairness in mind, as the composition of a *primary controller* and a set of *secondary controllers*, one for each monitored plant.

Both of the controller architectures temporarily disable interrupts to control overhead. One must therefore consider the impact of events missed during periods of non-monitoring on the monitoring results. The two applications of SMCO we consider are integer range analysis and the detection of under-utilized memory. For under-utilized memory detection, when an event is thrown, we are certain that the corresponding object is not stale. We can therefore ignore interrupts for a definite interval of time, without compromising soundness and at the same time lowering the monitoring percentage.

Similarly, for integer range analysis, two updates to an integer variable that are close to each other in time (e.g., consecutive increments to a loop variable) are often near each other in value as well. Hence, processing the interrupt for the first update and ignoring the second are often sufficient to accurately determine the variable's range, while also lowering monitoring percentage. For example, in the benchmarking experiments described in Sect. 4, we achieve high accuracy (typically 90% or better) in our integer range analysis with a target overhead of just 10%.

2.1 Target specification

The target specification for a single controlled plant is given as a timed language L, containing timed words of the form $a_1, t_1, \ldots, a_l, t_l$, where a_i is an event and t_i is the time at which a_i has occurred. Each plant has a local target monitoring percentage m_{lt}, which is effectively that plant's portion of the UTMP m_t. Specifically, L contains timed words $a_1, t_1, \ldots, a_l, t_l$ that satisfy the following conditions:

1. The average monitoring percentage $\overline{m} = (l\ p_a)\ /\ (t_l - t_1)$ is such that $\overline{m} \leq m_{lt}$, where p_a is the average time taken by the monitor and controller to process an event.
2. If the strict inequality $\overline{m} < m_{lt}$ holds, then the monitoring-percentage undershoot is due to time intervals with low activity during which all events are monitored.

The first condition bounds only the *mean monitoring percentage* $\overline{m}$ within a timed word $w \in L$. Hence, various policies for handling monitoring percentage, and thus enabling and disabling interrupts, are allowed. The second condition is a *best-effort* condition, which guarantees that if the target monitoring percentage is not reached, this is only because the plant does not throw enough interrupts. As our benchmarking results of Sect. 4 demonstrate, we designed the SMCO global and cascade controllers (described in Sect. 2.3) to satisfy these conditions.

When considering the target specification language L and the associated mean monitoring percentage $\overline{m}$, it is important to distinguish plants in which all interrupts can be disabled (as in Fig. 3) from the other (as in Fig. 4). Hardware-based execution platforms (e.g., CPU and MMU) and virtual machines such as the JVM belong to the former category.

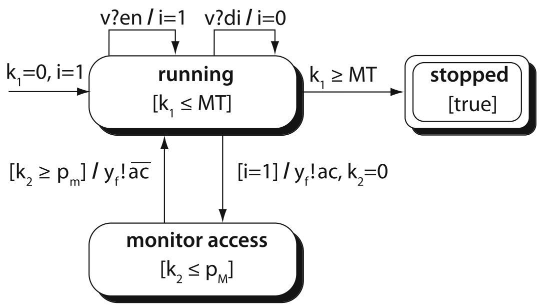


Fig. 3 Automaton for the hardware plant P of one monitored object

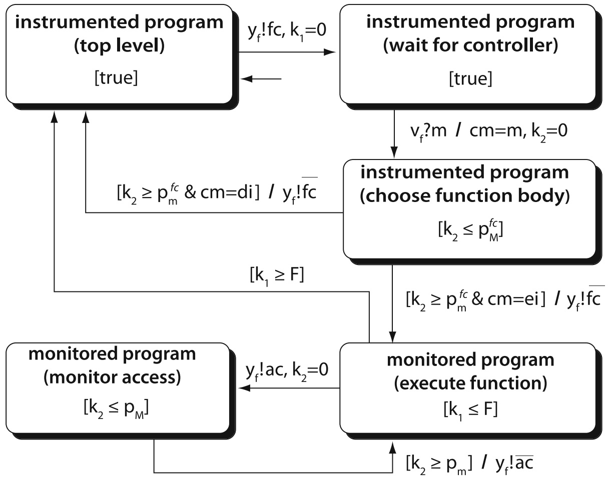


Fig. 4 Automaton for the software plant P of all monitored objects

(The JVM supports disabling of software-based interrupts through just-in-time compilation.)

Software plants written in C, however, typically belong to the latter category, because code inserted during instrumentation is not removed at runtime. In particular, as discussed in Sect. 2.2.2, when function calls are instrumented, the instrumented program always throws function-call interrupts a_{fc}. Consequently, for such plants, in addition to $\overline{m}$, there is also an unavoidable *base monitoring percentage* $m_b = k\ p_{fc}$, where k is the number of function calls.

2.2 The plant models

This section specifies the behavior of the above plant types in terms of *extended timed automata* (introduced below). For illustration purpose, each *hardware plant* is controlled by a secondary controller, and the unique *software plant* is controlled by the global controller.

2.2.1 The hardware plant

Timed automata (TA) [2] are finite-state automata extended with a set of clocks, the values of which are positive reals. Clock predicates on transitions are used to model timing behavior, while clock predicates appearing within locations (states) are used to enforce progress properties. Clocks may be reset by transitions. *Extended TA* are TA with local variables and a more expressive clock predicate/assignment language.

The hardware plant P is modeled by the extended TA in Fig. 3. Its alphabet consists of input and output events. The clock predicates labeling its locations and transitions are of the form $k \sim c$, where k is a *clock*, c is a natural number or variable, and $\sim$ is one of $<$, $\leq$, $=$, $\geq$ and $>$. For example, the predicate $k_1 \leq MT$ labeling P's state *running* is a clock constraint, where k_1 is a clock and MT is the maximum-monitoring time parameter discussed below.

Transition labels are of the form `[guard] In / Cmd`, where `guard` is a predicate over P's variables; `In` is a sequence of input events of the form $v?e$ denoting the receipt of value e (written as a pattern) on channel v; and `Cmd` is a sequence of output and assignment events. An output event is of the form $y!a$ denoting the sending of value a on channel y; an assignment event is simply an assignment of a value to a local variable of the automaton. All fields in a transition label are optional. The use of ? and ! to denote input and output events is standard and first appeared in Hoare's paper on CSP [12].

A transition is *enabled* when its guard is true and the specified input events (if any) have arrived. A transition is not *forced* to be taken, unless letting time flow would violate the condition (invariant) labeling the current location. For example, the transition out of the state *monitor access* in Fig. 3 is enabled as soon as $k_2 \geq p_m$, but not forced until $k_2 \geq p_M$. The choice is nondeterministic and allows to succinctly capture any transition in the interval $[p_m, p_M]$. This is a classic way of avoiding overspecification.

P has an input channel v where it may receive *enable* and *disable* commands, denoted en and di, respectively, and an output channel y_f where it may send messages for the beginning and ending of an *access*, denoted ac and $\overline{ac}$, respectively. Upon receipt of di, interrupt bit i is set to 0, which prevents the plant from sending further messages. Upon receipt of en, i is set to 1, which allows the plant to send an access message ac at arbitrary moments in time. Once an access message is sent, P resets the clock variable k_2 and transitions to a new state. At any time in the interval $[p_m, p_M]$, P can leave this state and send an end of access message $y_f!\overline{ac}$ to the controller.

P terminates when the maximum monitoring time MT, a parameter of the model, is reached, i.e., when clock k_1 reaches value MT. Initially, $i = 1$ and $k_1 = 0$.

A running program can have multiple hardware plants, with each plant a source of monitored events. For example, a program running under our NAP detection tool for finding under-utilized memory has one hardware plant for each monitored memory region. The NAP detector's controller can individually enable or disable interrupts for each hardware plant.

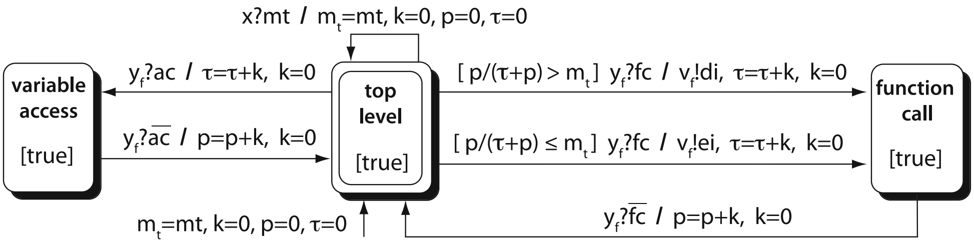


Fig. 5 Automaton for global controller

2.2.2 *The software plant*

In a software plant P, the application program is instrumented to handle, together with the monitor, the interrupt logic readily available to hardware plants (see Fig. 4).

A software plant represents a single function that can run with interrupts enabled or disabled. In practice, the function toggles interrupts by choosing between two copies of the function body each time it is called: one copy that is instrumented to send event interrupts and one that is left unmodified.

Whenever a function call happens at the *top level* state of P, the instrumented program resets the clock variable k_1, sends the message *fc* on y_f to the controller and *waits* for its response. If the response on v_f is *di*, indicating that interrupts are disabled, then the unmonitored version of the function body is called. This is captured in P by returning to the top level state at any time in the interval $[p_m^{fc}, p_M^{fc}]$. This interval represents the time required to implement the call logic.

If the response on v_f is *ei*, indicating that interrupts are enabled, then the monitored version of the function body is called. This is captured in P by transitioning to the state *execute function* within the same interval $[p_m^{fc}, p_M^{fc}]$.

Within the monitored function body, the monitor may send on y_f the beginning of access event *ac* to the controller, whenever a variable is accessed, and transition to the state *monitor access*. The time spent by monitoring this access is expressed with a transition back to *execute function* that happens at any time in the interval $[p_m, p_M]$. This transition sends an end of access message $\overline{ac}$ on y_f to the controller.

P terminates processing function f when the maximum monitoring time F, a parameter of the model, is reached; that is, when clock $k_1 \geq F$.

2.3 The controllers

2.3.1 *The global controller*

Integrative control uses previous behavior of the plant to control feedback. Integrative control has the advantage that it has good overall statistical performance for plants with consistent behavior and is relatively immune to *hysteresis*, in which periodicity in the output of the plant produces periodic, out-of-phase responses in the controller. Conversely, proportional control is highly responsive to changes in the plant's behavior, which makes it appropriate for long-running plants that exhibit change in behavior over time.

We have implemented an integral-like global controller for plants with consistent behavior. Architecturally, the global controller is in a feedback loop with a single plant representing all objects of interest to the runtime monitor.

The architecture of the global controller is thus exactly that of Fig. 1, which is identical to the classical plant-controller architecture of Fig. 2, except that in Fig. 1, the plant is decomposed into the runtime monitor and the software it is monitoring.

In presenting the extended TA for the global controller, we assume it is in a feedback loop with a software-oriented plant, the behavior of which is given by the extended TA of Fig. 4. This is done without loss of generality, as the global controller's state machine is simpler in the case of a hardware-oriented plant. The global controller thus assumes that the plant emits events of two types: *function-call events* and *access events*, where the former corresponds to the plant having entered a C function, and the latter corresponds to updates to integer variables, in the case of integer range analysis.

The global controller's automaton is given in Fig. 5 and consists of three locations: *top level*, the *function-call* processing location and the *variable-access* processing location. Besides the UTMP m_t, the automaton for the global controller makes use of the following variables: clock variable k, a running total τ of the program's execution time and a running total p of the instrumented program's observed processing time p. Variable τ keeps the time the controller spent in total (over repeated visits) in its top-level location, whereas variable p keeps the time the controller spent in total in its function-call and access-event processing locations. Hence, at every moment of time, the observed overhead is $o = p \,/\, \tau$ and the observed monitoring percentage is $m = p \,/\, (\tau + p)$.

In the top-level location, the controller can receive the UTMP on channel x. The controller transitions from the top-level to the function-call processing location whenever a function-call event occurs. In particular, when function f is called, the plant emits an *fc* signal to the controller along y_f (regardless of whether access event interrupts are enabled for f), transitioning the controller to the function-call processing location along one of two edges. If the observed

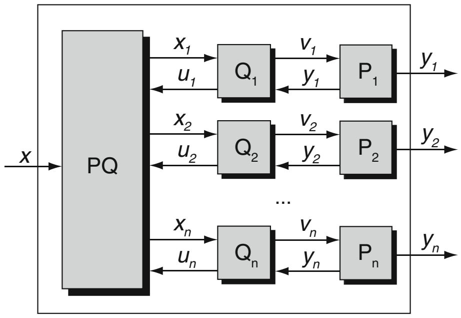


Fig. 6 Overall cascade control architecture

monitoring percentage for the entire program execution is above the UTMP m_t, the edge taken sends the di signal along v_f to disable monitoring of interrupts for that function call. Otherwise, the edge taken enables these interrupts. Thus, the global controller decides to enable/disable monitoring on a per-function call basis. Moreover, since the enable/disable decision depends on the sign of the *cumulative* error $e = m - m_t$, the controller is *integrative*.

The time taken in the function-call processing location, which the controller determines by reading clock k's value upon receipt of an $\overline{fc}$ signal from the plant is considered monitoring time; the transition back to the initial state thus adds this time to the total monitoring time p.

The controller transitions from the top-level to the variable- access processing location whenever a function f sends the controller an access event ac and interrupts are enabled for f. Upon receipt of an $\overline{ac}$ event signaling the completion of event processing in the plant, the controller measures the time it spent in its variable-access location and adds this quantity to p. To keep track of the plant's total execution time τ, each of the global controller's transitions exiting the initial location updates τ with the time spent in the top-level location.

Note that all of the global controller's transitions are event triggered, as opposed to time triggered, as it interacts asynchronously with the plant. This aspect of the controller model reflects the discrete event-based nature of our PID controllers.

2.3.2 The cascade controller

As per the discussion of monitoring percentage undershoot in Sect. 2.1, some plants (functions or objects in a C program) might *not* generate interrupts at a high rate, and therefore might not make use of the target monitoring percentage available to them. In such situations, it is desirable to redistribute such unused UTMP to more active plants, which are more likely to make use of this monitoring percentage. Moreover, this redistribution of the unused UTMP should be performed fairly, so that less active plants are not ignored.

This is the rationale for the SMCO cascade controller (see Fig. 6), which consists of a set of secondary controllers Q_i, each of which directly control a single plant P_i, and a primary controller PQ that controls the reference inputs x_i to the secondary controllers. Thus, in the case of cascade control, each monitored plant has its own secondary controller that enables and disables its interrupts. The primary controller adjusts the *local target monitoring percentage* (LTMP) m_{lt} for the secondary controllers.

The secondary controllers

Each monitored plant P has a secondary controller Q, the state machine for which is given in Fig. 7. Within each iteration of its main control loop, Q disables interrupts by sending message di along v upon receiving an access event ac along y, and subsequently enables interrupts by sending en along v. Consider the i-th execution of Q's control loop, and let τ_i be the *time monitoring is on* within this cycle; i.e., the time between events $v!en$ and $y?ac$. Let p_i be the time required to *process* event $y?ac$, and let d_i be the *delay time* until monitoring is restarted, i.e., until event $v!en$ is sent again. See Fig. 8 for a graphical depiction of these intervals.

Fig. 7 Automaton for secondary controller Q

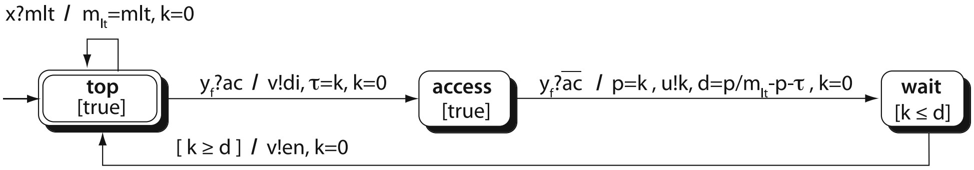
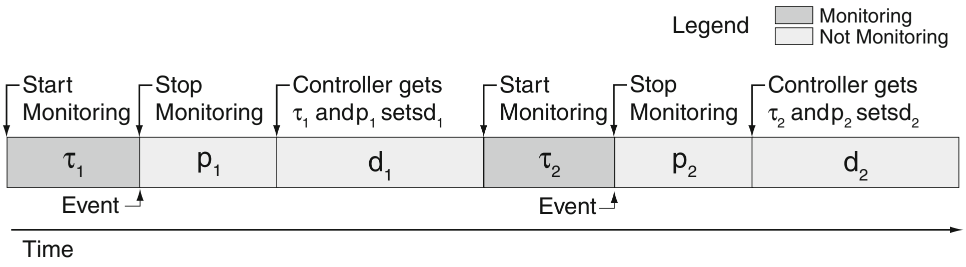

x?mlt / m_{lt}=mlt, k=0
top
[true]
y_f?ac / v!di, τ=k, k=0
access
[true]
$y_f?\overline{ac}$ / p=k , u!k, d=p/m_{lt}-p-τ , k=0
wait
[k ≤ d]
[k ≥ d] / v!en, k=0

Fig. 8 Timeline for secondary controller



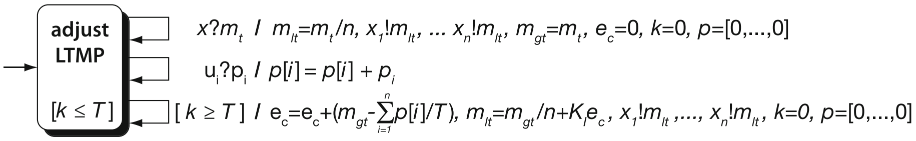

adjust
LTMP
$[k \leq T]$
$x?m_t$ / $m_{lt}=m_t/n$, $x_1!m_{lt}$, ... $x_n!m_{lt}$, $m_{gt}=m_t$, $e_c=0$, $k=0$, $p=[0,\ldots,0]$
$u_i?p_i$ / $p[i] = p[i] + p_i$
$[k \geq T]$ / $e_c=e_c+(m_{gt}-\sum_{i=1}^{n} p[i]/T)$, $m_{lt}=m_{gt}/n+K_I e_c$, $x_1!m_{lt}$,..., $x_n!m_{lt}$, $k=0$, $p=[0,\ldots,0]$

Fig. 9 Automaton for the primary controller

Then the overhead in the i-th cycle is $o_i = p_i \,/\, (\tau_i + d_i)$ and, accordingly, the *monitoring percentage* of the i-th cycle is $m_i = p_i \,/\, (p_i + \tau_i + d_i)$.

To ensure that $m_i = m_{lt}$ whenever the plant is throwing access events at a high rate, Q computes d_i as the least positive integer greater than or equal to $p_i/m_{lt} - (\tau_i + p_i)$. Choosing d_i this way lets the controller extend the total time spent in the i-th cycle so that its m_i is exactly the target m_{lt}.

To see how the secondary controller is like a proportional controller, regard p_i as a constant (p_i does not vary much in practice), so that p_i/m_{lt}—the desired value for the cycle time—is also a constant. The equation for d_i becomes now the difference between the desired cycle time (which we take to be the controller's reference value) and the actual cycle time measured when event i has completed processing. The value d_i is then equal to the proportional error for the i-th cycle, making the secondary controller behave like a proportional controller with proportional constant 1.

If plant P throws events at a low rate, then all events are monitored and $d_i = 0$. When processing of ac is completed, which is assumed to occur within the interval $[p_m, p_M]$, Q sends the processing time k to the primary controller along channel u.

The primary controller

Secondary controller Q achieves its LTMP m_{lt} only if plant P throws events at a sufficiently high rate. Otherwise, its mean monitoring percentage $\overline{m}$ is less than m_{lt}. When monitoring a large number of plants P_i simultaneously, it is possible to take advantage of this under-utilization of m_{lt} by increasing the LTMP of those controllers Q_i associated with plants that throw interrupts at a high rate. In fact, we can adjust the m_{lt} of *all* secondary controllers Q_i by the same amount, as the controllers Q_j of plants P_j with low interrupt rates will not take advantage of this increase. Furthermore, we do this every T seconds, a period of time we call the *adjustment interval*. The periodic adjustment of the LTMP is the task of the primary controller PQ.

Its extended TA is given in Fig. 9. After first inputting the UTMP m_t on x, PQ computes the initial LTMP to be m_t/n, thereby partitioning the global target monitoring percentage evenly among the n secondary controllers. It assigns this initial LTMP to the local variable m_{lt} and outputs it to the secondary controllers. It also assigns m_t to local variable m_{gt}, the *global target monitoring percentage* (GTMP). PQ also maintains an array p of total processing time, initially zero, such that $p[i]$ is the processing time used by secondary controller Q_i within the last adjustment interval of T seconds. Array entry $p[i]$ is updated whenever Q_i sends the processing time p_j of the most recent event a_j; i.e., $p[i]$ is the sum of the p_j that Q_i generates during the current adjustment interval.

When the time bound of T seconds is reached, PQ computes the error $e = m_{gt} - \sum_{i=1}^{n} p[i]/T$ as the difference between the GTMP and the observed monitoring percentage during the current adjustment interval. PQ also updates a cumulative error e_c, which is initially 0, such that $e_c = e_c + e$, making it the sum of the error over all adjustment intervals. To correct for the cumulative error, PQ computes an offset $K_I e_c$ that it uses to adjust m_{lt} down to compensate for over-utilization, and up to compensate for under-utilization. The new LTMP is set to $m_{lt} = m_{gt}/n + K_I e_c$ and sent to all secondary controllers, after which array p and clock k are reset.

Because the adjustment PQ makes to the LTMP m_{lt} over a given adjustment interval is a function of a cumulative error term e_c, primary controller PQ behaves as an *integrative controller*. In contrast, each secondary controller Q_i alone maintains no state beyond p_i and τ_i. They are therefore a form of *proportional controller*, which respond directly as the plant output changes. The controller parameter K_I in PQ's adjustment term $K_I e_c$ is known in control theory as the *integrative gain*. It is essentially a weight factor that determines to what extent the cumulative error e_c affects the local monitoring percentage m_{lt}. The larger the K_I value, the larger the changes PQ will make to m_{lt} during the current adjustment interval to correct for the observed overhead.

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

3 Design and implementation

This section presents two applications that we have implemented for SMCO. Section 3.1 describes our integer range analysis tool. Section 3.2 introduces our novel memory under-utilization monitor. Section 3.3 summarizes the development effort for these monitoring tools and their controllers.

3.1 Integer range analysis

Integer range analysis [7] determines the range (minimum and maximum value) of each integer variable in the monitored execution. These ranges are useful for finding program

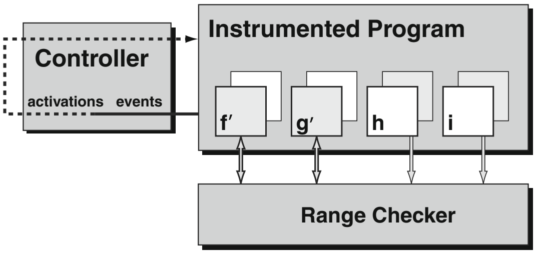


Fig. 10 SMCO architecture for `range-solver`

errors. For example, analyzing ranges on array subscripts may reveal bounds violations.

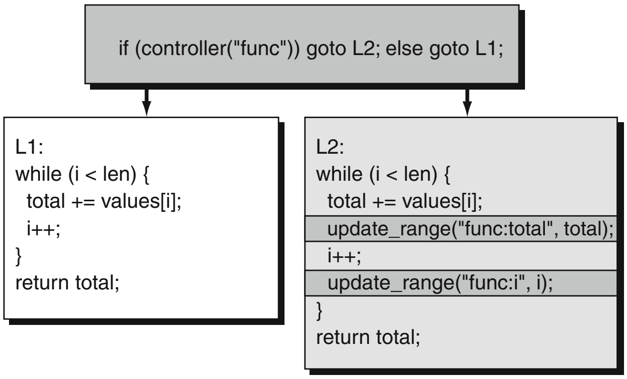
Figure 10 is an overview of `range-solver`, our integer range-analysis tool. `range-solver` uses *compiler-assisted instrumentation* (CAI), an instrumentation framework based on a plug-in architecture for GCC [5]. Our `range-solver` plug-in adds range-update operations after assignments to global, function-level static, and stack-scoped integer variables. The Range Checker module (shown in Fig. 10) consumes these updates and computes ranges for all tracked variables. Range updates are enabled or disabled on a per-function basis. In Fig. 10, monitoring is enabled for functions f and g; this is reflected by the instrumented versions of their function bodies, labeled f' and g', appearing in the foreground.

To allow efficient enabling and disabling of monitoring, the plug-in creates a copy of the body of every function to be instrumented, and adds instrumentation only to the copy. A distributor block at the beginning of the function calls the SMCO controller to determine whether monitoring for the function is currently enabled. If so, the distributor jumps to the instrumented version of the function body; otherwise, control passes to the original, unmodified version. Fig. 11 shows a function modified by the `range-solver` plug-in to have a distributor block and a duplicate instrumented function body. Functions without integer updates are not duplicated and always run with monitoring off.

Because monitoring is enabled or disabled at the function level, the instrumentation notifies the controller of function-call events. As shown in Fig. 10, the controller responds by activating or deactivating monitoring for instrumented functions. With the global controller, there is a single "on–off" switch that affects all functions: when monitoring is off, the uninstrumented versions of all function bodies are executed. The cascade controller maintains a secondary controller for each instrumented function and can switch monitoring on and off for individual functions.

Timekeeping

As our controller logic relies on measurements of monitoring time, `range-solver` queries the system time whenever it makes a control decision. The `RDTSC` instruction is the fastest and most precise timekeeping mechanism on the x86 platform. It returns the CPU's timestamp counter (*TSC*), which stores a processor cycle timestamp (with sub-nanosecond resolution), without an expensive system call.



```
if (controller("func")) goto L2; else goto L1;
```

```
L1:
while (i < len) {
  total += values[i];
  i++;
}
return total;
```

```
L2:
while (i < len) {
  total += values[i];
  update_range("func:total", total);
  i++;
  update_range("func:i", i);
}
return total;
```

Fig. 11 `range-solver` adds a distributor with a call to the SMCO controller. The distributed passes control to either the original, uninstrumented function body shown on the *left*, or the instrumented copy shown on the *right*

However, we found that even `RDTSC` can be too slow for our purposes. On our testbed, we measured the `RDTSC` instruction to take 45 cycles on average, more than 20 times longer than an arithmetic instruction. With time measurements necessary on every function call for our `range-solver`, this was too expensive. Our first `range-solver` implementation called `RDTSC` inline for every time measurement, resulting in a 23% overhead even with all monitoring turned off.

To reduce the high cost of timekeeping, we modified the `range-solver` to spawn a separate "clock thread" to handle its timekeeping. The clock thread periodically calls `RDTSC` and stores the result in a memory location that `range-solver` uses as its clock. `range-solver` can read this clock with a simple memory access. This is not as precise as calling `RDTSC` directly, but it is much more efficient.

3.2 Memory under-utilization

Our memory under-utilization detector is designed to identify allocated memory areas ("allocations" for short) that are not accessed during time periods, the duration of which equals or exceeds a user-specified threshold. We refer to such a time period as a non-accessed period, or NAP, and to the user-specified threshold as the *NAP threshold*. Figure 12 depicts accesses to an allocation and the resulting NAPs. For example, the time from access b to access c exceeds the NAP threshold, so it is a NAP. Note that we are not detecting allocations that are never touched (i.e., leaks), but rather allocations that are not touched for a sufficiently long period of time to raise concerns about memory usage efficiency.

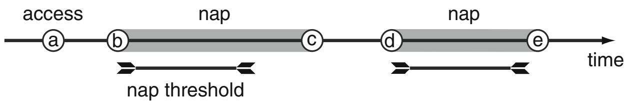


Fig. 12 Accesses to an allocation, and the resulting NAPs. NAPs can vary in length, and multiple NAPs can be reported for the same allocation

Sampling by enabling and disabling monitoring at the function level would be ineffective for finding NAPs. An allocated region could appear to be in a NAP if some functions that access it are not monitored, resulting in false positives. Ideally, we want to enable and disable monitoring *per allocation*, so that we do not unintentionally miss accesses to monitored allocations. To achieve per-allocation control over monitoring, we introduce a novel memory-access interposition mechanism that takes advantage of memory-protection hardware. Figure 13 shows the architecture of our NAP detector. To enable monitoring for an allocation, the controller calls `mprotect` to turn on read and write protections for the memory page containing the allocation.

In Fig. 13, memory regions m and o are protected: any access to them will cause a page fault. The NAP detector intercepts the page fault first to record the access; information about each allocation is stored in a splay tree. Then, the controller interprets the fault as an event and turns monitoring off for the faulting allocation by removing its read and write access protections. Because the NAP detector restores read and write access after a fault, the faulting instruction can execute normally once the page fault handler returns. It is not necessary to emulate the effects of a faulting instruction.

Within the cascade controller, each allocation has a secondary controller that computes a delay d, after which time monitoring should be re-enabled for the allocation. After processing an event, the controller checks for allocations that are due for re-enabling. If no events occur for a period of time, a background thread performs this check instead. The background thread is also responsible for periodically checking for memory regions that have been protected for longer than the NAP threshold and reporting them as NAPs.

We did not integrate the NAP detector with the global controller because its means of overhead control does not allow for this kind of per-allocation control. When a monitored event e occurs and the accumulated overhead exceeds o_t, a global controller should establish a period of *global* zero overhead by disabling all monitoring. Globally disabling monitoring has the same problem as disabling monitoring at the function level: allocations that are unlucky enough to be accessed only during these zero-overhead periods will be erroneously classified as NAPs.

We also implemented a shared library that replaces the standard memory-management functions, including `malloc` and `free`, with versions that store information about the creation and deletion of allocations in splay trees.

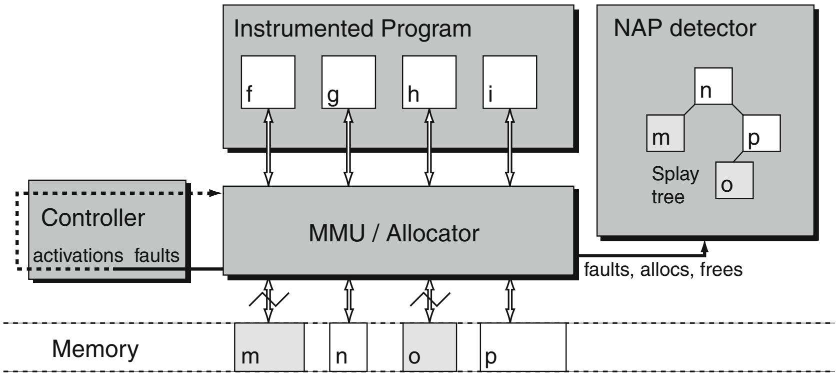
To control access protections for individual allocations, each such allocation must reside on a separate hardware page, because `mprotect` has page-level granularity. To reduce the memory overhead, the user can set a size cutoff: allocations smaller than the cutoff are not monitored, so they do not need to take up an entire page (under-utilized small allocations are of less interest than under-utilized large allocations anyway). In our experiments, we chose double the page size (equal to 8 KB on our test machine) as the cutoff, limiting the maximum memory overhead to 50%. Though this cutoff would be high for many applications, in our experiments, 75% of all allocations were monitored.

3.3 Implementation effort

To implement and test the `range-solver` tool described in Sect. 3.1, we developed two libraries, totaling 821 lines of code, which manage the logic for the cascade and global controllers and perform the integer range analysis. We also developed a 1,708-line GCC plug-in that transforms functions to report integer updates to our integer range-analysis library.

The NAP detector described in Sect. 3.2 consists of a 2, 235-line library that wraps the standard memory-allocation functions, implements the cascade controller logic and transparently handles page faults in the instrumented program.

Fig. 13 SMCO architecture for NAP detector



4 Evaluation

This section describes a series of benchmarks that together show that SMCO fulfills its goals: it closely adheres to the specified target overhead, allowing the user to specify a precise trade-off between overhead and monitoring effectiveness. In addition, our cascade controller apportions the target overhead to all sources of events, ensuring that each source gets its fair share of monitoring time.

Our results highlight the difficulty inherent in achieving these goals. The test workloads vary in behavior considerably over the course of an execution, making it impractical to predict sources of overhead. Even under these conditions, SMCO is able to control the observed overhead fairly well.

To evaluate SMCO's versatility, we tested it on two workloads, one CPU-intensive and one I/O-intensive, and with our two different runtime monitors. Section 4.1 discusses our experimental testbed. Section 4.2 describes the workloads and profiles them in order to examine the challenges involved in controlling monitoring overhead. In Sect. 4.3, we benchmark SMCO's ability to control the overhead of our integer range analysis monitor using both of our control strategies. Section 4.4 benchmarks SMCO overhead control with our NAP detector. Section 4.5 explains how we optimized certain controller parameters.

4.1 Testbed

Since controlling overhead is most important for long-running server applications, we chose a server-class machine for our testbed. Our benchmarks ran on a Dell PowerEdge 1950 with two quad-core 2.5 GHz Intel Xeon processors, each with a 12 MB L2 cache and 32 GB of memory. It was equipped with a pair of Seagate Savvio 15K RPM SAS 73 GB disks in a mirrored RAID. We configured the server with 64-bit CentOS Linux 5.3, using a CentOS-patched 2.6.18 Linux kernel.

For our observed overhead benchmark figures, we averaged the results of ten runs and computed 95% confidence intervals using Student's t-distribution. Error bars represent the width of a measurement's confidence interval.

We used the `/proc/sys/vm/drop_caches` facility provided by the Linux kernel to drop page, inode and dentry caches before each run of our I/O-intensive workload to ensure cold caches and to prevent consecutive runs from influencing each other.

4.2 Workloads

We tested our SMCO approach on two applications: the CPU-intensive `bzip2` and an I/O-intensive `grep` workload. The `bzip2` benchmark is a data compression workload from the SPEC CPU2006 benchmark suite, which is designed to maximize CPU utilization [11]. This benchmark uses the `bzip2` utility to compress and then decompress a 53 MB file consisting of text, JPEG image data and random data.

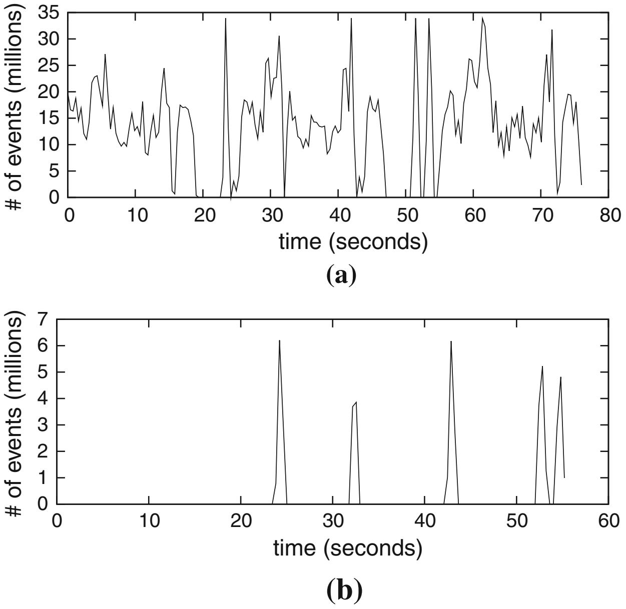


Fig. 14 Event distribution histogram for the most updated variable (**a**) and 99th most updated variable (**b**) in `bzip2`. Execution time (x-axis) is split into 0.4-s buckets. The y-axis shows the number of events in each time bucket

Our `range-solver` monitor, described in Sect. 3.1, found 80 functions in `bzip2`, of which 61 contained integer assignments and 445 integer variables, 242 of which were modified during execution. Integer update events were spread very unevenly among these variables. The least updated variables were assigned only one or two times during a run, while the most updated variable was assigned 2.5 billion times.

Figure 14 shows the frequency of accesses to two different variables, the most updated variable and the 99th most updated variable, over time. The data were obtained by instrumenting `bzip2` to monitor a single specified variable with unbounded overhead. The monitoring runs for these two variables took 76.4 and 55.6 s, respectively. The two histograms show different extremes: the most updated variable is constantly active, while accesses to the 99th most updated variable are concentrated in short periods of high activity. Both variables, however, experience heavy bursts of activity that make it difficult to predict monitoring overhead.

Our I/O-intensive workload uses GNU `grep` 2.5, the popular Linux regular expression search utility. In our benchmarks, `grep` searches the entire GCC 4.5 source tree (about 543 MB in size) for an uncommon pattern. When we tested the workload with the Unix `time` utility, it reported that these runs typically used only 10–20% CPU time. Most of each run was spent waiting for read requests, making this an I/O-heavy workload. Because the `grep` workload repeats the same short tasks, we found that its variable accesses were distributed more uniformly than in `bzip2`. Our `range-solver` reported 489 variables, with 128 actually updated in each run, and 149 functions, 87 of which contained integer assignments. The most updated variable was assigned 370 million times.

4.3 Range solver

We benchmarked the `range-solver` monitor discussed in Sect. 3.1 on both workloads using both of the controllers in Sect. 2. Sections 4.3.1 and 4.3.2 present our results for the global controller and cascade controller, respectively. Section 4.3.3 compares the results from the two controllers. Section 4.3.4 discusses `range-solver`'s memory overhead.

4.3.1 Global controller

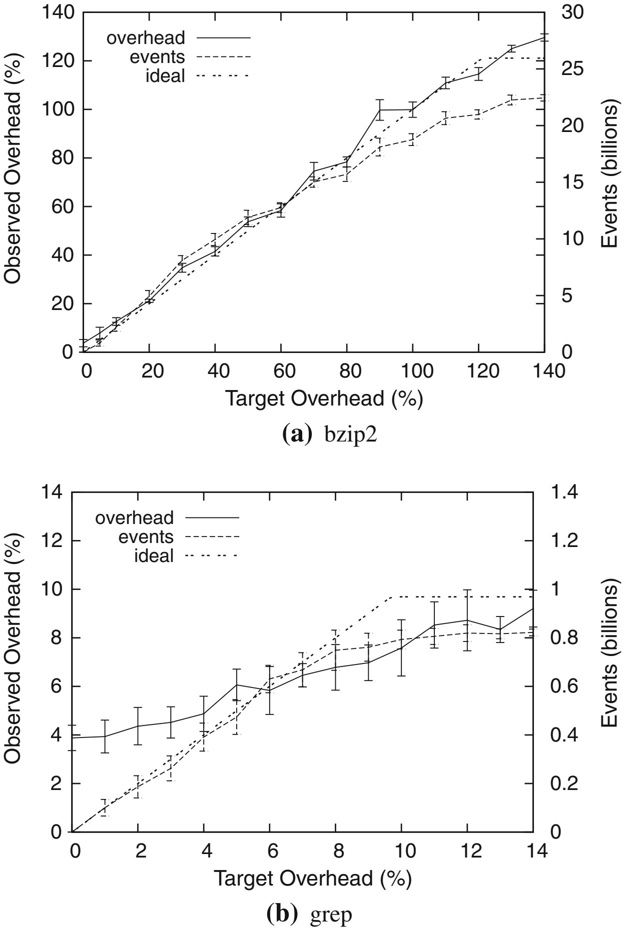
Figure 15 shows how the global controller performs on our workloads for a range of target overheads (on the x-axis), with the observed overhead on the y-axis and the total number of events monitored on the $y2$-axis for each target overhead setting. With target overhead set to 0%, both workloads ran with an actual overhead of 4%, which is the controller's *base overhead*. The base overhead is due to the controller logic and the added complexity from unused instrumentation.

The dotted line in each plot shows the ideal result: observed overhead equals target overhead up to an ideal maximum. We computed the ideal maximum to be the observed overhead from monitoring all events in the program with all control turned off. Any observed overhead above the ideal maximum is the result of overhead incurred by the controller.

At target overheads of 10% and higher, Fig. 15a shows that the global controller tracked the specified target overhead all the way up to 140% in the `bzip2` workload. The `grep` workload (Fig. 15b) showed a general upward trend for increasing target overheads, but never exceeded 9% observed overhead. In fact, at 9% overhead, `range-solver` is already at nearly full coverage, with 99.7% of all program events being monitored. The `grep` workload's low CPU usage imposes a hard limit on `range-solver`'s ability to use overhead. The controller has no way to exceed this limit. Confidence intervals for the `grep` workload were generally wider than for `bzip2`, because I/O operations are noisier than CPU operations, making runtimes less consistent.

4.3.2 Cascade controller

Figure 16 shows results from experiments that are the same as those for Fig. 15 except using the cascade controller instead of the global controller. The results were similar. On the `bzip2` workload, the controller tracked the target overhead well from 10% to 100%. With targets higher than 100%, the observed overhead continued to increase, but the controller was not able to adjust the overhead high enough to reach the target, because the observed overhead was already close to the 120% maximum. On the `grep` workload, we saw the same upward trend and eventual saturation with 9% observed overhead monitoring 99.5% of events.



(a) bzip2

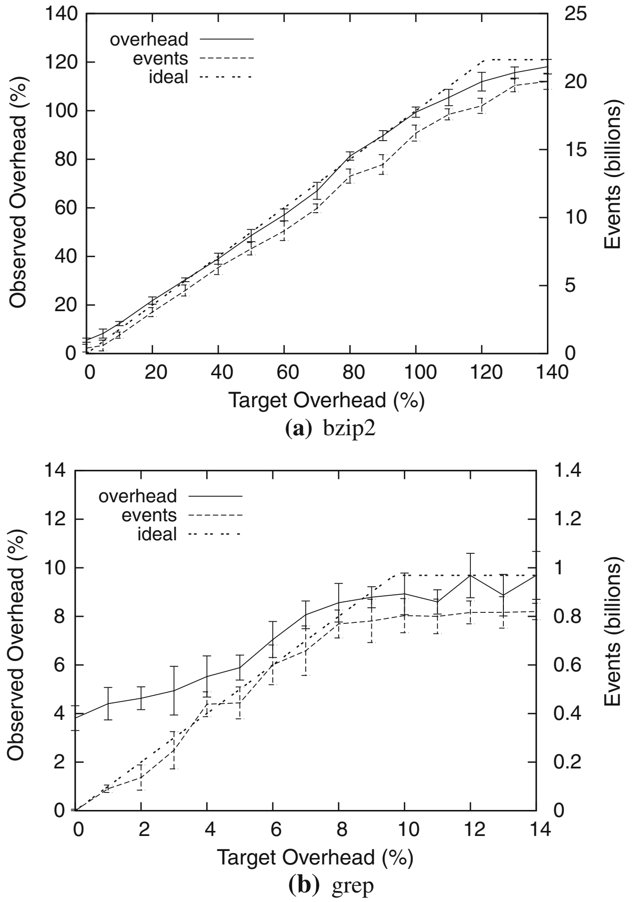
(b) grep

Fig. 15 *Global controller with range-solver* observed overhead (y-axis) for a range of target overhead settings (x-axis) and two workloads

4.3.3 Controller comparison

The global and cascade controllers differ in the distribution of overhead across different event sources. To compare them, we developed an accuracy metric for the results of a bounded-overhead range-solver run. We measured the accuracy of a bounded-overhead run of the range-solver against a reference run with full coverage of all variables (allowing unbounded overhead). The reference run determined the actual range for every variable.

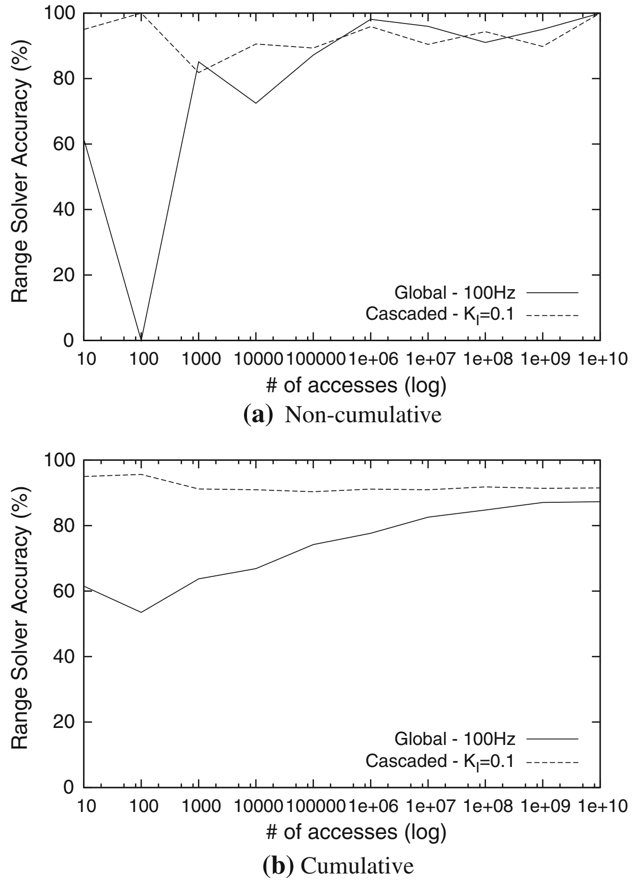
In a bounded-overhead run, the accuracy of a range computed for a single variable is the ratio of the computed range size to the range's actual size (which is known from the reference run). Missed updates in a bounded-overhead run can



(a) bzip2

(b) grep

Fig. 16 *Cascade controller with range-solver* observed overhead (y-axis) for a range of target overhead settings (x-axis) and two workloads



(a) Non-cumulative

(b) Cumulative

Fig. 17 *Comparison of range-solver accuracy* for both controllers with `bzip2` workload. Variables are grouped by total number of updates

cause `range-solver` to report smaller ranges, so this ratio is always in the interval [0, 1]. For a set of variables, the accuracy is the average accuracy for all variables in the set.

Figure 17 shows a breakdown of `range-solver`'s accuracy on how frequently variables are updated. We grouped variables into sets with geometrically increasing bounds: the first set containing variables with 1–10 updates, the second group containing variables with 10–100 updates, etc. Figure 17a shows the accuracy for each of these sets, and Fig. 17b shows the cumulative accuracy, with each set containing the variables from the previous set.

We used 10% target overhead for these examples, because we believe that low target overheads represent the most likely used cases. However, we found similar results for all other target overhead values that we tested.

The cascade controller's notion of fairness results in better coverage, and thus better accuracy, for rarely updated variables. In this example, the cascade controller had better accuracy than the global controller for variables with fewer than 100 updates. As the global controller does not seek to fairly distribute overhead to these variables, it monitored a smaller percentage of their updates. Most dramatically, Fig. 17a shows that the global controller had 0 accuracy for all variables in the 10–100 updates range, meaning it did not monitor more than one event for any variable in that set. The three variables in the workload with 10–100 updates were used while there was heavy activity, causing their updates to get lost in the periods when the global controller had to disable monitoring to reduce overhead.

However, with the same overhead, the global controller was able to monitor many more events than the cascade controller, because it did not spend time executing the cascade controller's more expensive secondary controller logic. These extra events gave the global controller much better coverage for frequently updated variables. Specifically, it had better accuracy for variables with more than 10^6 updates.

Between these two extremes, i.e., for variables with 100–10^6 updates, both approaches had similar accuracy. The cumulative accuracy in Fig. 17b shows that overall, considering all variables in the program, the two controllers achieved similar accuracy. The difference is primarily in where the accuracy was distributed.

Table 1 `range-solver` memory usage, including executable size, virtual memory usage (VSZ) and physical memory usage (RSS)

	Exe size (KB)	VSZ	RSS (KB)
bzip2 (unmodified)	68.6	213 KB	207
bzip2 (global)	262	227 KB	203
bzip2 (cascade)	262	225 KB	201
grep (unmodified)	89.2	61.4 MB	1,260
grep (global)	314	77.1 MB	1,460
grep (cascade)	314	78.2 MB	1,470

4.3.4 Memory overhead

Although `range-solver` does not use SMCO to control its memory overhead, we measured memory use of our controllers for both workloads. Table 1 shows our memory overhead results. Here *Exe Size* is the size of the compiled binary after stripping debugging symbols (as is common in production environments). This size includes the cost of the SMCO library, which contains the compiled controller and monitor code. *VSZ* is the total amount of memory mapped by the process, and *RSS* (resident set size) is the total amount of that virtual memory stored in RAM. We obtained the peak VSZ and RSS for each run using the Unix `ps` utility.

Both binaries increased in size by three to four times. Most of this increase is the result of function duplication, which at least doubles the size of each instrumented function. Duplicated functions also contain a distributor block and instrumentation code. The 17 KB SMCO library adds a negligible amount to the instrumented binary's size. As few binaries are more than several megabytes in size, we believe that even a 4× increase in executable size is acceptable for most environments; this is more true these days, with increasing amounts of RAM in popular 64-bit systems.

The worst case increase in virtual memory use was only 27.4%, for the `grep` workload with the cascade controller. The additional virtual memory is allocated statically to store integer variable ranges and per-function overhead measurements (when the cascade controller is used). This extra memory scales linearly with the number of integer variables and functions in the monitored program, and not with runtime memory usage. The `bzip2` workload uses more memory than `grep`, so we measured in this case a much lower 6.6% virtual memory overhead.

4.4 NAP detector

Figure 18 shows the results of our NAP detector, described in Sect. 3.2, on `bzip2` using the cascade controller. The NAP detector's instrumentation incurs no overhead while ignoring events, so it has a very low base overhead of only 1.1%.

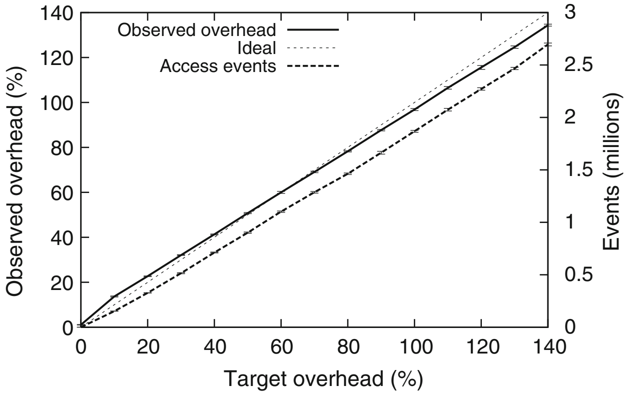


Fig. 18 *Cascade controller with NAP detector* observed overhead (y-axis) and number of monitored memory access events ($y2$-axis) for a range of target overhead settings (x-axis) running on the `bzip2` workload

Table 2 NAP detector memory usage, including executable size, virtual memory usage (VSZ), and physical memory usage (RSS)

	Exe size (KB)	VSZ (KB)	RSS (KB)
bzip2 (unmodified)	68.6	213	207
bzip2 (cascade)	89.4	225	209

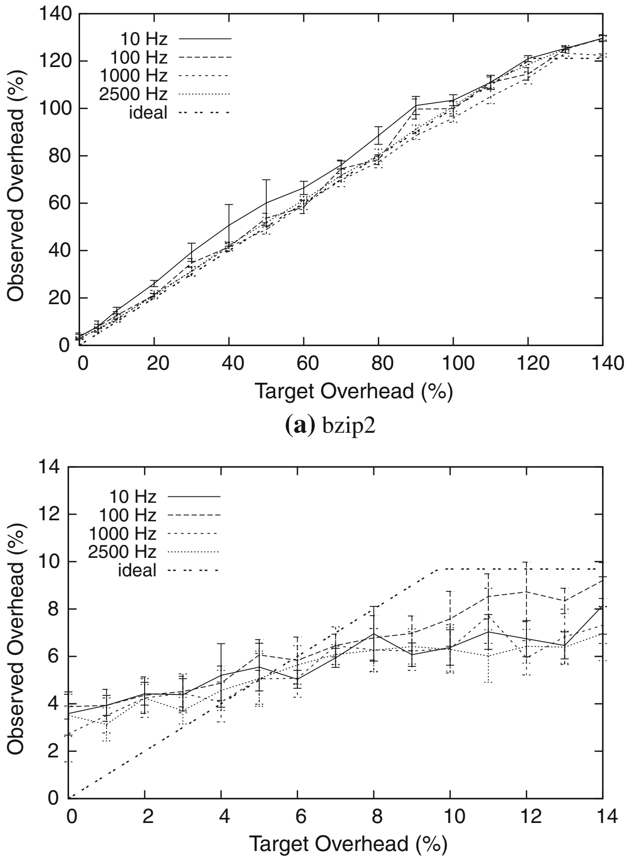
The NAP detector also tracks the user-specified target overhead well from 10–140%. The results also show that the NAP detector takes advantage of extra overhead that the user allows it. As target overhead increased, the number of monitored events scaled smoothly from only 300,000 to over 4 million.

Because the global controller is not designed to use per-allocation sampling (as explained in Sect. 3.2), we used only cascade control for our NAP detector experiments.

Table 2 shows memory overhead for the NAP detector. *Exe Size*, *VSZ* and *RSZ* are the same as in Sect. 4.3.4. Although it is possible for the NAP detector to increase memory usage by as much as 50% (see Sect. 3.2), the `bzip2` benchmark experienced only a 5.3% increase in virtual memory usage (VSZ). All of the allocations monitored in the benchmark were 10–800 times larger than the page size; so, forcing these allocations to be in separate pages resulted in very little wasted space. The 20.8 KB increase in Exe size is from the statically linked SMCO library.

4.5 Controller optimization

This section describes how we chose values for several of our control parameters to get the best performance from our controllers. Section 4.5.1 discusses our choice of clock precision for time measurements. Section 4.5.2 explains how we chose the integrative gain and adjustment interval for the primary



(a) bzip2

(b) grep

Fig. 19 *Observed overhead for global controller clock frequencies* with four different clock frequencies and two workloads using `range-solver` instrumentation

controller. Section 4.5.3 discusses optimizing the adjustment interval for an alternate primary controller.

4.5.1 Clock frequency

The control logic for our `range-solver` implementation uses a *clock thread*, as described in Sect. 3.1, which trades off precision for efficiency. This thread can keep more precise time by updating its clock more frequently, but more frequent updates result in higher timekeeping overhead. Recall that the `RDTSC` instruction is relatively expensive, taking 45 cycles on our testbed.

We performed experiments to determine how much precision was necessary to control overhead accurately. Figure 19 shows the `range-solver` benchmark in Fig. 15 repeated for four different clock frequencies. The *clock frequency* is how often the clock thread wakes up to read the TSC.

At only 10 Hz, the controller's time measurements were not accurate enough to keep the overhead below the target. With infrequent updates, most monitoring operations occurred without an intervening clock tick and therefore appeared to take 0 seconds. The controller ended up with an underestimate of the actual monitoring overhead and thus overshot its goal.

At 100 Hz, however, controller performance was good and the clock thread's impact on the system was still negligible, incurring the 45-cycle `RDTSC` cost only 100 times for every 2.5 billion processor cycles on our test system. More frequent updates did not perform any better and wasted resources, so we chose 100 Hz for our clock update frequency.

In choosing the clock frequency, we wanted to ensure that we also preserved SMCO's effectiveness. Figure 20 shows the accuracy of the `range-solver` at 10% target overhead using the four clock frequencies we tested. We used the same accuracy metric as in Sect. 4.3.3 and plotted the results as in Fig. 17. The `range-solver` accuracy is similar for 100, 1,000 and 2,500 Hz clocks. These three values resulted in similar observed overheads in Fig. 19. It therefore makes sense that they achieve similar accuracy, since SMCO is designed to support trade-offs between effectiveness and overhead. The 10 Hz run has the best accuracy, but this result is misleading because it attains that accuracy at the cost of higher overhead than the user-requested 10%. Testing these clock frequencies at higher target overheads showed similar behavior. Note that the 100 Hz curves in Fig. 20 are the same as the global controller curves from the controller comparison in Fig. 17.

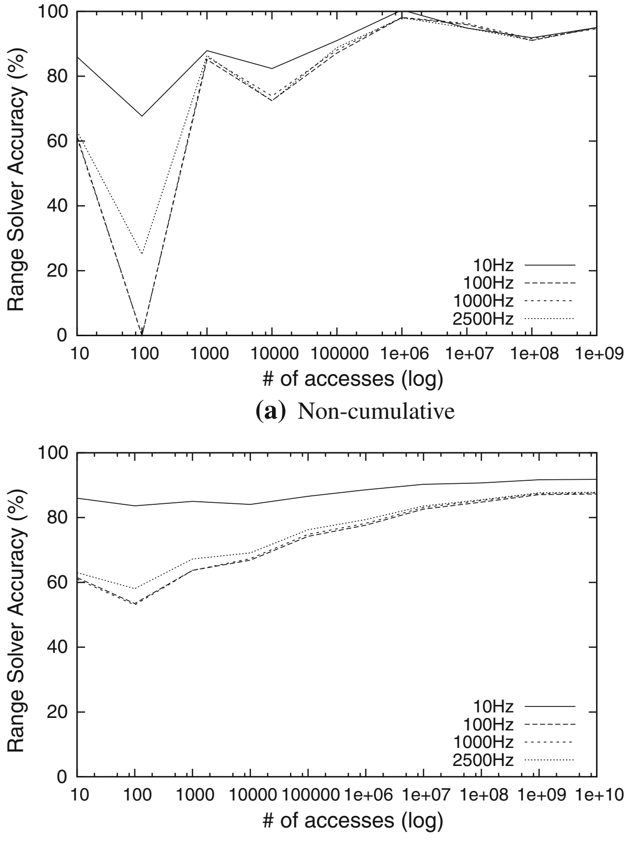
4.5.2 Integrative gain

The primary controller component of the cascade controller discussed in Sect. 2 has two parameters: the integrative gain K_I and the adjustment interval T. The integrative gain is a weight factor that determines how much the cumulative error e_c (the deviation of the observed monitoring percentage from the target monitoring percentage) changes the local target monitoring percentage m_{lt} (the maximum percent of execution time that each monitored plant is allowed to use for monitoring). When K_I is high, the primary controller makes larger changes to m_{lt} to correct for observed deviations.

The adjustment interval is the period of time between primary controller updates. With a low T value, the controller adjusts m_{lt} more frequently.

There are processes for choosing good control parameters for most applications of control theory, but our system is tolerant enough that good values for K_I and T can be determined experimentally.

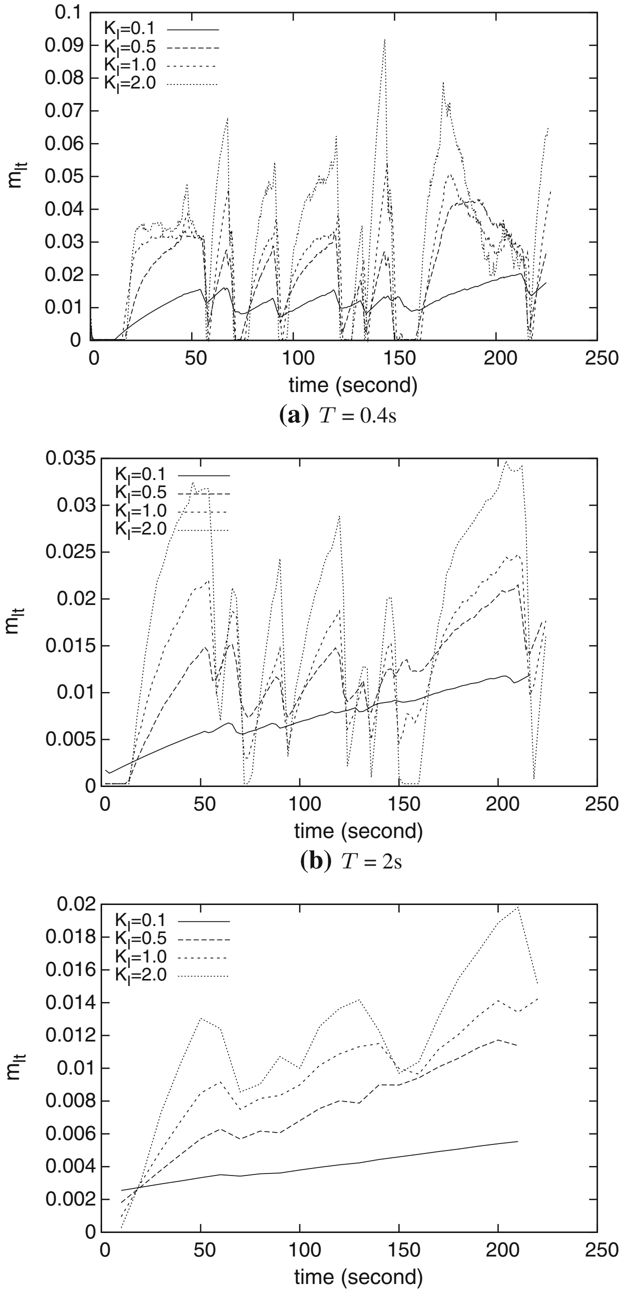
We tuned the controller by running `range-solver` on an extended `bzip2` workload with a range of values for K_I and T and then recording how the primary controller's output variable, m_{lt}, stabilized over time. Figure 21 shows our results for four K_I values and three T values with target overhead set to 20% for all runs. These results revealed trends in the effects of adjusting the controller parameters.



(a) Non-cumulative

(b) Cumulative

Fig. 20 *Accuracy of global controller clock frequencies* with four different clock frequencies using `range-solver` instrumentation. Variables are grouped by total number of updates



(a) T = 0.4s

(b) T = 2s

(c) T = 10s

Fig. 21 *Local target monitoring percentage* (m_{lt}) *over time* during `bzip2` workload for `range-solver` with cascade control. Results shown with target overhead set to 20% for four different values of K_I and three values of T

In general, we found that higher values of K_I increased controller responsiveness, allowing m_{lt} to more quickly compensate for under-utilization of monitoring time, but at a cost of driving higher-amplitude oscillations in m_{lt}. This effect is most evident with $T = 0.4$ s (see Fig. 21a). All the values we tested from $K_I = 0.1$ to $K_I = 2.0$ successfully met our overhead goals, but values greater than 0.1 oscillated wildly enough that the controller had to sometimes turn monitoring off completely (by setting m_{lt} to almost 0) to compensate for previous spikes in m_{lt}. With $K_I = 0.1$, however, m_{lt} oscillated stably for the duration of the run after a 50-s warm-up period.

When we changed T to 2 s (see Fig. 21b), we observed that 0.1 was no longer the optimal value for K_I. But with $K_I = 0.5$, we were able to obtain performance as good as the optimal performance with T =0.4 s: the controller met its target overhead goal and had the same 50-s warm-up time. We do see the consequences of choosing too small a K_I; however, with $K_I = 0.1$, the controller was not able to finish warming up before the benchmark finished, and the system failed to achieve its monitoring goal.

The same problem occurred when $T = 10$ s (see Fig. 21c): the controller updated so infrequently that it only completed its warm-up for the highest K_I value we tried. Even with the highest K_I, the controller still undershot its monitoring percentage goal.

Because of its stability, we chose $K_I = 0.1$ (with $T = 0.4$ s) for all of our cascade-controlled `range-solver`

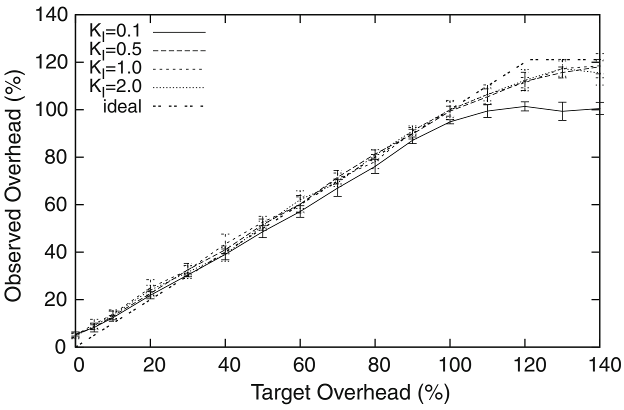


Fig. 22 *Observed overhead for primary controller K_I values* using `range-solver` with $T = 400$ ms and four different K_I values

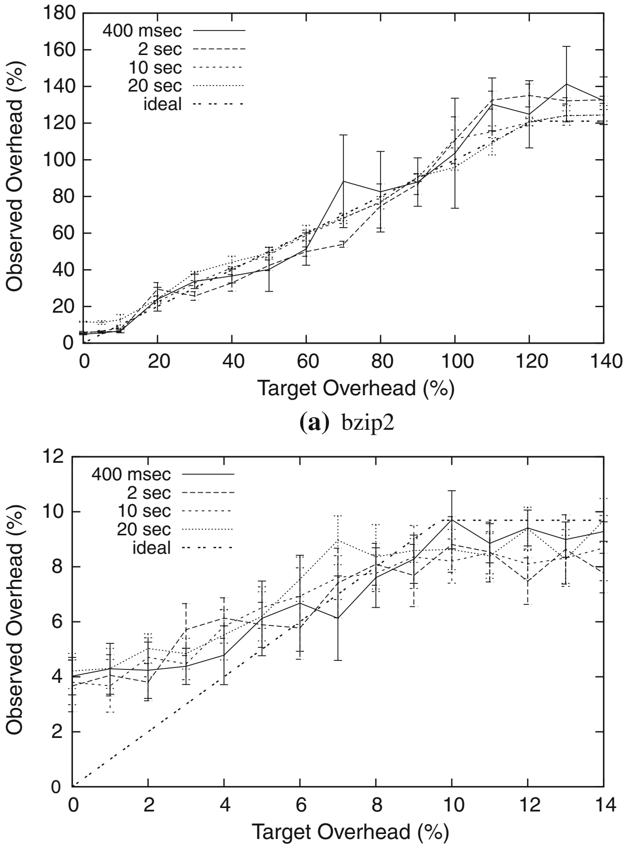
experiments with low target overhead. Figure 22 shows how the controller tracked target overhead for all the K_I values we tried. Although $K_I = 0.1$ worked well for low overheads, we again observed warm-up periods that were too long when target overhead was very high. The warm-up period took longer with very high overhead because of the larger gap between target overhead and the initial observed overhead. To deal with this discrepancy, we actually use two K_I values, $K_I = 0.1$ for normal cases, and a special high-overhead K_I^H for cases with target overhead greater than 70%. We chose $K_I^H = 0.5$ using the same experimental procedure we used for K_I.

4.5.3 Adjustment interval

Before settling on the integrative control approach that we use for our primary controller, we attempted an ad hoc control approach that yielded good results for the `range-solver`, but was not stable enough to control the NAP detector. We also found the ad hoc primary controller to be more difficult to adjust than the integrative controller. Though we no longer use the ad hoc approach, the discussion of how we tuned it illustrates some of the properties of our system.

Rather than computing error as a difference, the ad hoc controller computes error e_f fractionally as the global target monitoring percentage (GTMP, as in Sect. 2.3.2) divided by the observed global monitoring percentage for the entire program run so far. The value of e_f is greater than one when the program is under-utilizing its monitoring time and less than one when the program is using too much monitoring time. After computing the fractional error, the controller computes a new value for m_{lt} by multiplying it by e_f.

The ad hoc primary controller has only one parameter to adjust. Like the integrative controller, it has an interval time T between updates to m_{lt}.



(a) bzip2

(b) grep

Fig. 23 *Observed overhead for an ad hoc cascade controller's T values* with four different values of *T* and two `range-solver` workloads

Figure 23 shows the `range-solver` benchmark in Fig. 16 repeated for four values of T. For the `bzip2` workload (Fig. 16a), the best results were obtained with a T interval of 10 s. Smaller T values led to an unstable primary controller: the observed overhead varied from the target and results became more random, as shown by the wider confidence intervals for these measurements.

This result contradicts the intuition that a smaller T should stabilize the primary controller by allowing it to react faster. In fact, the larger T gives the controller more time to correctly observe the trend in utilization among monitoring sources. For example, with a short T, the first adjustment interval of `bzip2`'s execution used only a small fraction of all the variables in the program. The controller quickly adjusted m_{lt} very high to compensate for the monitoring time that the unaccessed variables were failing to use. In subsequent intervals, the controller needed to adjust m_{lt} down sharply to offset the overhead from many monitoring sources becoming active at once. The controller spent the rest of its run oscillating to correct its early error, never reaching equilibrium during the entire run.

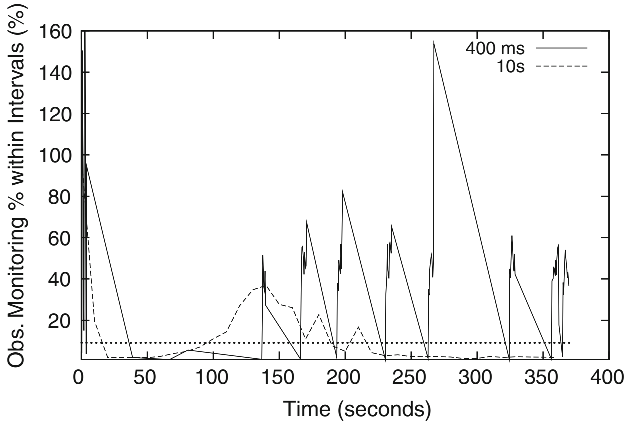


Fig. 24 *Observed monitoring percentage over bzip2* `range-solver` *execution.* The percent of each adjustment interval spent monitoring for two values of T. The target monitoring percentage is shown as a *dotted horizontal line*

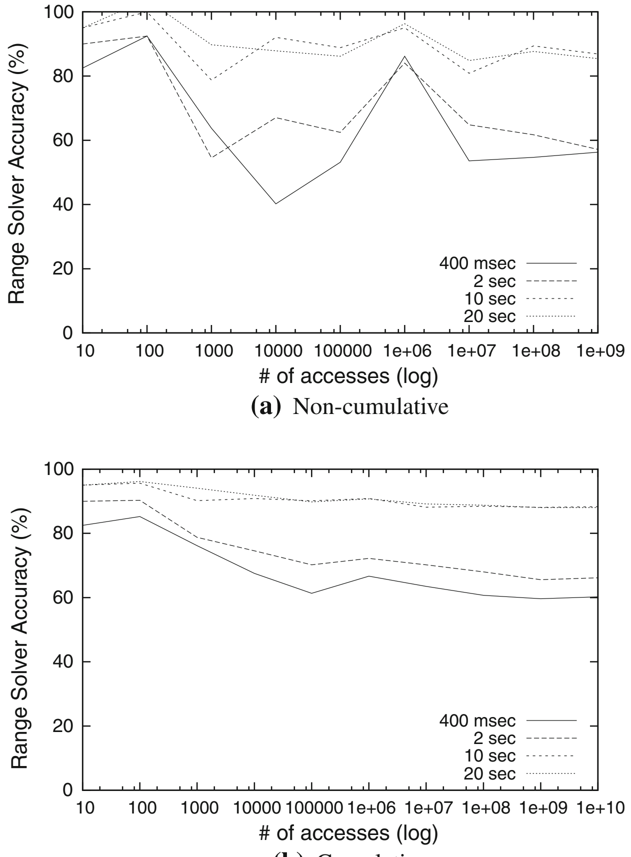
Figure 24 shows how the observed monitoring percentage for each adjustment interval fluctuates during an extended `range-solver` run of the `bzip2` workload as a result of the primary controller's m_{lt} adjustments. With $T = 0.4$ s, the observed monitoring percentage spikes early on, so the observed overhead is actually very high at the beginning of the program. Near 140 s, the controller overreacts to a change in program activity, causing another sharp spike in observed monitoring percentage. As execution continues, the observed monitoring percentage sawtooths violently, and there are repeated bursts of time when the observed percentage is much higher than the target percentage (meaning observed overhead is much higher than the user's target overhead).

With $T = 10$ s, the observed monitoring percentage still fluctuates, but the extremes do not vary as far from the target. As execution continues, the oscillations dampen, and the system reaches stability.

In our `bzip2` workload, the first few primary controller intervals were the most critical: bad values at the beginning of execution were difficult to correct later on. The more reasonable 10-s T made its first adjustment after a larger sample of program activity, so it did not overcompensate. Overall, we expect that a primary controller with a longer T is less likely to be misled by short bursts or lulls in activity.

There is a practical limit to the length of T, however. In Fig. 23a, a controller with $T = 20$ s overshot its target overhead. Because the benchmark runs for only about 1 min, the slower primary controller was not able to adjust m_{lt} often enough to converge on a stable value before the benchmark ended.

As in Sect. 4.5.1, we also tested how the choice of T affects the `range-solver`'s accuracy, using the same accuracy metric as in Sect. 4.3.3. Figure 25 shows the accuracy for the `bzip2` workload using four different values for T with a 10% target overhead. The accuracy results confirm our choice of 10 s. Only the 20-s value for T yields better accuracy, but it does so because it consumes more overhead than the primary controller with $T = 10$ s. Tests with higher target overheads gave similar results.



(a) Non-cumulative

(b) Cumulative

Fig. 25 *Accuracy of cascade controller T values* with four values of T on the `bzip2` workload using `range-solver` instrumentation. Variables are grouped by total number of updates

Evaluation summary

Our results show that in all the cases we tested, SMCO was able to track the user-specified target overhead for a wide range of target overhead values. Even when there were too few events during an execution to meet the target overhead goal, as was the case for the `grep` workload with target overheads greater than 10%, SMCO's controller was able to achieve the maximum possible observed overhead by monitoring nearly all events. We also showed that the overhead trade-off is a useful one: higher overheads allowed for more effective monitoring. These results are for challenging workloads with unpredictable bursts in activity.

Although our results relied on choices for several parameters, we found that it was practical to find good values for all of these parameters empirically. As future work, we plan to explore procedures for automating the selection of optimal controller parameters, which can vary with different types of monitoring.

5 Related work

Chilimbi and Hauswirth [10] propose an overhead control mechanism for memory under-utilization detection that approximates memory-access sampling by monitoring a program's basic blocks for memory accesses. Low overhead is achieved 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. As a non-accessed period (NAP) is most commonly associated with a memory allocation and not a basic block, there is no clear association between the sampling rate for blocks and confidence that some memory region is under-utilized. In contrast, SMCO's NAP detector uses virtual memory hardware to directly monitor areas of allocated memory, and overhead control is realized by enabling and disabling the monitoring of memory areas appropriately.

Artemis [8] reduces CPU overhead due to runtime monitoring by enabling monitoring for only certain function executions. By default, Artemis monitors a function execution only if the function is called in a *context* that it has not seen before. In theory, a function's context consists of the values of all memory locations it accesses. Storing and comparing such contexts would be too expensive, so Artemis actually uses weaker definitions of "context" and "context matching," which may cause it to miss some interesting behaviors [8]. Artemis' context-awareness is orthogonal to SMCO's feedback control approach. SMCO could be augmented to prefer new contexts, as Artemis does, in order to monitor more contexts within an overhead bound. The main difference between Artemis and SMCO is in the goals they set for overhead management. Artemis uses context awareness to reduce—but not bound—the overhead. The overhead from monitoring a function in every context varies during program execution and can be arbitrarily high, especially early in the execution, when most contexts have not yet been seen. The user does not know in advance whether the overhead will be acceptable. In contrast, SMCO is designed to always keep overhead within a user-specified bound. As a result of this different goal, SMCO uses very different techniques than Artemis. In summary, overhead reduction techniques like those in Artemis are useful but, unlike SMCO, they do not directly address the problem of unpredictable and unacceptably high overheads.

Liblit et al.'s [13] statistical debugging technique seeks to reduce per-process monitoring overhead by partitioning a monitoring task across many processes running the same program.

This approach is attractive when applicable, but has several limitations: it is less effective when an application is being tested and debugged by a small number of developers; privacy concerns may preclude sharing of monitoring results from deployed applications; and for some monitoring tasks, it may be difficult to partition the task evenly or correlate monitoring data from different processes. Furthermore, in contrast to SMCO, statistical debugging does not provide the ability to control overhead based on a user-specified target level.

We first introduced the concept of a user-specified target overhead, along with the idea of using feedback control to enforce that target, in an NSF-sponsored workshop [4]. The controller we implemented in that workshop is the basis for our cascade controller.

The Quality Virtual Machine (QVM) [3] also supports runtime monitoring with a user-specified target overhead. Its overhead-management technique is similar to our cascade controller in that it splits control into global and local overhead targets and prioritizes infrequently executed probes to increase coverage. The QVM overhead manager assigns a *sampling frequency* to each overhead source, and it adjusts the sampling frequency to control overhead. Unlike SMCO, which uses an integral controller to adjust local target monitoring percentages, these adjustments are made by an *ad hoc* controller that lacks theoretical foundations. The section on the overhead manager in QVM does not describe (or justify) the computations used to adjust sampling frequency.

In its benchmarks, QVM did not track overhead goals as accurately as SMCO. For example, configured for 20% target overhead, two of QVM's benchmarks, `eclipse` and `fop`, showed an actual overhead of about 26% [3]. Though we cannot test Java programs, our `bzip` workload is a comparable CPU-bound benchmark. SMCO's worst case for `bzip` was a 22.7% overhead with NAP detector instrumentation, shown in Fig. 18. This suggests that controllers, like SMCOs, which are based on established control theory principle can provide more accurate control. This is not surprising, since meeting an overhead goal is inherently a feedback control problem.

SMCO achieves accurate overhead control even when monitored events happen very frequently. Our `range-solver` attempts to track every integer assignment in a program, handling many more events than QVM's monitors, which track calls to specified methods. We measured the rate of total events per second in benchmarks from both systems: the rate of integer assignment events in our `bzip2` workload and the rate of potentially monitored method calls in the DaCapo benchmarks that QVM uses. The `bzip2` event rate was more than 50 times the DaCapo rate. Our technique of toggling monitoring at the function level, using function duplication, makes it possible to cope with high event rates by reducing the number of controller decisions. Even with this technique, we found that direct calls to `RDTSC` by the controller—an approach that is practical for the low event rates in QVM's benchmarks—are too expensive at high event rates like those in `range-solver`. We overcame this problem by introducing a separate clock thread, as described in Sect. 3.1.

QVM and SMCO make sampling decisions at different granularities, reflecting their orientation toward monitoring different kinds of properties. QVM makes monitoring decisions at the granularity of objects. In theory, for each object, QVM monitors all relevant operations (method invocations) or none of them. In practice, this is problematic, because the target overhead may be exceeded if QVM decides to track an object that turns out to have a high event rate. QVM deals with this by using *emergency shutdown* to abort monitoring of such objects. In contrast, SMCO is designed for monitors that do not need to observe every operation on an object to produce useful results. For example, even if monitoring is temporarily disabled for a memory allocation, our NAP detector can resume monitoring the allocation and identify subsequent NAPs.

QVM and SMCO are designed for very different execution environments. QVM operates in a modified Java Virtual Machine (JVM). This makes implementation of efficient monitoring considerably easier, because the JVM sits conveniently between the Java program and the hardware. SMCO, on the other hand, monitors C programs, for which there is no easy intermediate layer in which to implement interception and monitoring. Monitoring of C programs is complicated by the fact that C is weakly typed, pointers and data can be manipulated interchangeably, and all memory accesses are effectively global: any piece of code in C can potentially access any memory address via any pointer. Low-level instrumentation techniques allow SMCO to control overhead in spite of these complications: function duplication reduces overhead from instrumentation that is toggled off, and our novel use of memory management hardware allows efficient tracking of accesses to heap objects.

6 Conclusions and future work

We have presented software monitoring with controllable overhead (SMCO), an approach to overhead control for the runtime monitoring of software. SMCO is optimal in the sense that it monitors as many events as possible without exceeding the target overhead level. This is distinct from other approaches to software monitoring that promise low or adaptive overhead, but where overhead, in fact, varies per application and under changing usage conditions. The key to SMCO's performance is the use of underlying control strategies for a nonlinear control problem represented in terms of the composition of timed automata.

Using SMCO as a foundation, we developed two sophisticated monitoring tools: an integer range analyzer, which uses code-oriented instrumentation, and a NAP detector, which uses memory-oriented instrumentation. Both the per-function checks in the integer range analyzer and the per-memory-area checks in the NAP detector are activated and deactivated by the same generic controller, which achieves a user-specified target overhead with either of these systems running.

Our extensive benchmarking results demonstrate that it is possible to perform runtime monitoring of large software systems with fixed target overhead guarantees. As such, SMCO is promising both for developers, who desire maximal monitoring coverage, and system administrators, who need a way to effectively manage the impact of runtime monitoring on system performance. Moreover, SMCO is fully responsive to both increases and decreases in system load, even highly bursty workloads, for both CPU- and I/O-intensive applications. As such, administrators need not worry about unusual effects in instrumented software caused by load spikes.

Future work

There are many potential uses for SMCO, including such varied techniques as lockset profiling and checking, runtime type checking, feedback-directed algorithm selection and intrusion detection. SMCO could be used to manage disk or network time instead of CPU time. For example, a background file-system consistency checker could use SMCO to ensure that it gets only a specific fraction of disk time, and a background downloader could use SMCO to ensure that it consumes only a fixed proportion of network time.

Though our cascade controller adheres to its target overhead goals very well, it is dependent on somewhat careful adjustment of the K_I control parameter. Changes to the monitor can alter the system enough to require retuning K_I. It would be useful to automate the process for optimizing K_I for low and high overheads so that developing new monitors does not require tedious experimentation. An automated procedure could model the response time and level of oscillation as a function of K_I in order to find a good trade-off.

We also believe that SMCO as a technique can be improved. One improvement would be to handle *dependencies* between monitored events in cases where correctly monitoring an event requires information from previous events. For example, if a runtime type checker fails to monitor the event that initializes an object's type, it would be useless to monitor type-checking events for that object; the controller should therefore be able to ignore such events.

Acknowledgments The authors would like to thank the anonymous reviewers for their invaluable comments and suggestions. They would also like to thank Michael Gorbovitski for insightful discussions and help with the benchmarking results. Research was supported in part by AFOSR Grant FA9550-09-1-0481, NSF Grants CNS-0509230, CCF-0926190, and CNS-0831298 and ONR Grant N00014-07-1-0928.

References

1. Aziz, A., Balarin, F., Brayton, R.K., Dibenedetto, M.D., Sladanha, A., Sangiovanni-Vincentelli, A.L.: Supervisory control of finite state machines. In: Wolper, P. (ed.) 7th International Conference On Computer Aided Verification, vol. 939, pp. 279–292. Springer, Liege, Belgium (1995)
2. Alur, R., Dill, D. L.: A theory of timed automata. Theoret. Comput. Sci. **126**(2), 183–235 (1994)
3. Arnold, M., Vechev, M., Yahav, E.: QVM: An efficient runtime for detecting defects in deployed systems. In: Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). ACM, Nashville, TN (2008)
4. Callanan, S., Dean, D.J., Gorbovitski, M., Grosu, R., Seyster, J., Smolka, S.A., Stoller, S.D., Zadok, E.: Software monitoring with bounded overhead. In: Proceedings of the 2008 NSF Next Generation Software Workshop, in conjunction with the 2008 International Parallel and Distributed Processing Symposium (IPDPS 2008), Miami (2008)
5. Callanan, S., Dean, D.J., Zadok, E.: Extending GCC with modular GIMPLE optimizations. In: Proceedings of the 2007 GCC Developers' Summit, Ottawa (2007)
6. Cantrill, B., Shapiro, M.W., Leventhal, A.H.: Dynamic instrumentation of production systems. In: Proceedings of the Annual USENIX Technical Conference, pp. 15–28 (2004)
7. Fei, L., Midkiff, S.P.: Artemis: Practical runtime monitoring of applications for errors. Technical report TR-ECE-05-02. Electrical and Computer Engineering, Purdue University. http://docs.lib.purdue.edu/ecetr/4/ (2005)
8. Fei, L., Midkiff, S.P.: 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 (2006)
9. Franklin, G.F., Powell, J.D., Workman, M.: Digital Control of Dynamic Systems, Third Edition. Addison Wesley Longman, Inc, Boston (1998)
10. Hauswirth, M., Chilimbi, T.M.: Low-overhead memory leak detection using adaptive statistical profiling. In: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2004), pp. 156–164 (2004)
11. Henning, J.L.: SPEC CPU2006 benchmark descriptions. Comput. Archit. News **34**(4), 1–17 (2006)
12. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**, 666–677 (1978)
13. Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I.: Bug isolation via remote program sampling. In: Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '03), San Diego (2003)
14. Moore, R.: A universal dynamic trace for Linux and other operating systems. In: Proceedings of the 2001 USENIX Annual Technical Conference (2001)
15. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event systems. SIAM J. Control Optim. **25**(1), 206–230 (1987)
16. Ramadge, P.J., Wonham, W.M.: Supervisory control of timed discrete-event systems. IEEE Trans. Autom. Control **38**(2), 329–342 (1994)
17. Seward, J., Nethercote, N., Fitzhardinge, J.: Valgrind. http://valgrind.kde.org (2004)
18. Wang, Q.-G., Ye, Z., Cai, W.-J., Hang, C.-C.: PID control for multivariable processes. Lecture Notes in Control and Information Sciences. Springer, Berlin (2008)
19. Wong-Toi, H., Hoffmann, G.: The control of dense real-time discrete event systems. In: Proceeedings of 30th Conference on Decision and Control, pp. 1527–1528, Brighton (1991)