Runtime Verification for High-Confidence Systems: A Monte Carlo Approach

Sean Callanan Radu Grosu Abhishek Rai Scott A. Smolka
Mike R. True Erez Zadok

Computer Science Department
Stony Brook University

Appears in the proceedings of the Second Workshop on Model-Based Testing (MBT 2006)

Abstract

We present a new approach to runtime verification that utilizes classical statistical techniques such as Monte Carlo simulation, hypothesis testing, and confidence interval estimation. Our algorithm, MCM, uses *sampling-policy automata* to vary its sampling rate dynamically as a function of the current confidence it has in the correctness of the deployed system. We implemented MCM within the Aristotle tool environment, an extensible, GCC-based architecture for instrumenting C programs for the purpose of runtime monitoring. For a case study involving the dynamic allocation and deallocation of objects in the Linux kernel, our experimental results show that Aristotle reduces the runtime overhead due to monitoring, which is initially high when confidence is low, to levels low enough to be acceptable in the long term as confidence in the monitored system grows.

1 Introduction

In previous work [7], we presented the $\mathtt{MC}^2$ algorithm for *Monte Carlo Model Checking*. Given a (finite-state) reactive program P, a temporal property φ, and parameters ϵ and δ, $\mathtt{MC}^2$ samples up to M random executions of P, where M is a function of ϵ and δ. Should a sample execution reveal a counter-example, $\mathtt{MC}^2$ answers false to the model-checking problem $P \models \varphi$. Otherwise, it decides with *confidence* $1 - \delta$ and *error margin* ϵ, that P indeed satisfies φ. Typically the number M of executions that $\mathtt{MC}^2$ samples is much smaller than the actual number of executions of P. Moreover, each execution sampled starts in an initial state of P, and terminates after a finite number of execution steps, when a cycle in the state space of P is reached.

In this paper, we show how the technique of Monte Carlo model checking can be extended to the problem of *Monte Carlo monitoring and runtime ver-*

ification. Our resulting algorithm, `MCM`, can be seen as a runtime adaptation of `MC`^{2}, one whose dynamic behavior is defined by *sampling-policy automata* (SPA). Such automata encode strategies for dynamically varying `MCM`'s sampling rate as a function of the current confidence in the monitored system's correctness. A sampling-policy automaton may specify that when a counterexample is detected at runtime, the sampling rate should be increased since `MCM`'s confidence in the monitored system is lower. Conversely, if after M samples the system is counterexample-free, the sampling rate may be reduced since `MCM`'s confidence in the monitored system is greater.

The two key benefits derived from an SPA-based approach to runtime monitoring are the following:

- As confidence in the deployed system grows, the sampling rate decreases, thereby mitigating the overhead typically associated with long-term runtime monitoring.
- Because the sampling rate is automatically increased when the monitored system begins to exhibit erroneous behavior (due either to internal malfunction or external malevolence), Monte Carlo monitoring dynamically adapts to internal mode switches and to changes in the deployed system's operating environment.

A key issue addressed in our extension of Monte Carlo model checking to the runtime setting is: What constitutes an adequate notion of a *sample*? In the case of Monte Carlo runtime verification, the monitored program is already deployed, and restarting it after each sample to return the system to an initial state is not a practical option. Given that every reactive system is essentially a sense-process-actuate loop, in this paper we propose weaker notions of initial state that are sufficient for the purpose of dynamic sampling. One such notion pertains to the manipulation of instances of dynamic types: Java classes, dynamic data structures in C, etc. In this setting, a sample commences in the program state immediately preceding the allocation of an object o and terminates in the program state immediately following the deallocation of o, with these two states being considered equivalent with respect to o.

To illustrate this notion of runtime sampling, we consider the problem of verifying the safe use of *reference counts* (RCs) in the Linux *virtual file system* (VFS). The VFS is an abstraction layer that permits a variety of separately-developed file systems to share caches and present a uniform interface to other kernel subsystems and the user. Shared objects in the VFS have RCs so that the degree of sharing of a particular object can be measured. Objects are placed in the reusable pool when their RCs go to zero, objects with low RCs can be swapped out, but objects with high RCs should remain in main memory. Proper use of RCs is essential to avoid serious correctness and performance problems for all file systems.

To apply Monte Carlo runtime monitoring to this problem, we have defined Real Time Linear Temporal Logic formulas that collectively specify what it means for RCs to be correctly manipulated by the VFS. We further implemented the MCM algorithm within the Aristotle environment for Monte Carlo monitoring. Aristotle provide a highly extensible, GCC-based architecture for instrumenting C programs for the purposes of runtime monitoring. Aristotle realizes this architecture via a simple modification of the GNU C compiler (GCC) that allows one to load an arbitrary number of plug-ins dynamically and invoke code from those plug-ins at the tree-optimization phase of compilation.

Using a very simple sampling policy, our results show that Aristotle brings runtime overhead, which is initially very high when confidence is low, down to long-term acceptable levels. For example, a benchmark designed to highlight overheads under worst-case conditions exhibited a 10x initial slowdown; 11 minutes into the run, however, we achieved 99.999% confidence that the error rate for both classes of reference counts was below one in 10^5. At this point, monitoring for that class was reduced, leaving an overhead of only 33% from other monitoring.

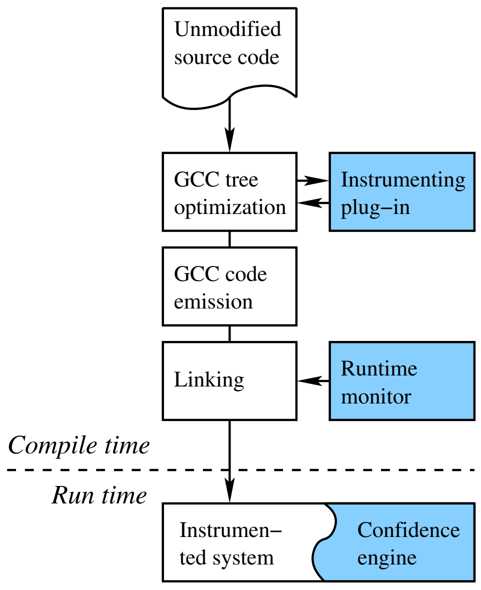
In addition to reference counts, Aristotle currently provides Monte Carlo monitoring support for the correct manipulation of pointer variables (bounds checking), lock-based synchronization primitives, and memory allocation library calls. Due to its extensible architecture based on plug-ins, support for other system features can be easily added.

The rest of the paper is organized as follows. Section 2 describes our system design. Section 3 presents our Monte Carlo runtime monitoring algorithm. Section 4 details the Aristotle design and implementation. Section 5 gives an example application of Aristotle, and Section 6 discusses related work. Section 7 contains our concluding remarks and directions for future work.

2 Aristotle Design Overview

Figure 1 depicts the various stages of operation for Aristotle as it processes a system's source code. A modified version of the GNU C compiler (GCC) parses the source code, invoking an *instrumenting plug-in* to process the control flow graph for each function. The instrumenting plug-in inserts calls to verification code at each point where an event occurs that could affect the property being checked. The verification code is part of a *runtime monitor*, which maintains auxiliary runtime data used for property verification and is bound into the software at link time.

The runtime monitor interacts with the *confidence engine*, which implements a sampling policy based on our Monte Carlo runtime monitoring algorithm (described in Section 3). The confidence engine maintains a confidence level for the properties being checked and may implement a sampling policy automaton to regulate the instrumentation or perform other actions. This regulation can be based on changes in the confidence level and could respond to other events in the system, such as the execution of rarely-used code paths.



In an RC debugger:	**In a bounds checker:**
Instruments all code that manipulates reference counts	Instruments all memory accesses and allocations
Monitors objects; flags improper use and leaks	Monitors areas currently allocated; flags bad accesses
Regulates monitoring based on sampled behavior and sampling policy	

Fig. 1. Architectural overview of the Aristotle system.

3 Monte Carlo Monitoring

In this section, we present our `MCM` algorithm for Monte Carlo monitoring and runtime verification. We first present `MCM` in the context of monitoring the correct manipulation of reference counts (RCs) in the Linux virtual file system (VFS). RCs are used throughout the Linux kernel, not only to prevent premature deallocation of objects, but also to allow different subsystems to indicate interest in an object without knowing about each other's internals. Safe use of reference counts is an important obligation of all kernel subsystems. We then consider generalizations of the algorithm to arbitrary dynamic types.

In the case of the Linux VFS, the objects of interest are *dentries* and *inodes*, which the VFS uses to maintain information about file names and data blocks, respectively. The VFS maintains a static pool of these objects and uses RCs for allocation and deallocation purposes: a free object has an RC of zero and may be allocated to a process; an object with a positive RC is considered in-use and may only be returned to the free pool when the state of the RC returns to zero. Additionally, an object with a high reference count is less likely to be swapped out to disk.

To apply Monte Carlo runtime monitoring to this problem, we first define the properties of interest. These are formally defined in Figure 2.

Each of these properties is formalized using *Real-Time Linear Temporal Logic* [2], where `G`, `F` and `X` are unary temporal operators. `G` requires the sub-formula over which it operates to be true *Globally* (in all states of an execution), `F` requires it to hold *Finally* (in some eventual state of an execution), and `X` requires it to hold *neXt* (in the next state of an execution). Also, an unprimed variable refers to its value in the current state and the primed version refers to its value in the next state. Each property uses universal

(stI)	$\forall o : C.\ \mathtt{G}\ o.\mathrm{rc} \geq 0$	RC values are always non-negative.
(trI)	$\forall o : C.\ \mathtt{G}\ \lvert o'.\mathrm{rc} - o.\mathrm{rc} \rvert \leq 1$	RC values are never incremented or decremented by more than 1.
(lkI)	$\forall o : C.\ \mathtt{G}\ o'.\mathrm{rc} \neq o.\mathrm{rc} \Rightarrow \mathtt{XF}_{\leq T}\ o'.\mathrm{rc} \leq o.\mathrm{rc}$	A change in the value of an RC is always followed within time T by a decrement.

Fig. 2. Reference-count correctness properties.

quantification over all instances o of a dynamic type C.

The first property is a *state invariant* (stI) while the second property is a *transition invariant* (trI). The third property is a *leak invariant* (lkI) that is intended to capture the requirement that the RC of an actively used object eventually returns to zero. It is expressed as a time-bounded liveness constraint, with time bound T.

Since each of these properties can be proved false by examining a finite execution, they are safety properties, and one can therefore construct a deterministic finite automaton (DFA) A that recognizes violating executions [10,16]. The synchronous composition (product) C_A of C with A is constructed by instrumenting C with A such that C violates the property in question iff an object o of type C can synchronize with A so as to lead A to an accepting state.

We view an object o of type C as executing in a closed system consisting of the OS and its environment. We assume that the OS is deterministic but the environment is a (possibly evolving) Markov chain; i.e., its transitions may have associated probabilities. As a consequence, C_A is also a Markov chain. Formally, a *Markov chain* $M = (X, E, p, p_0)$ consists of a set X of *states*; a set $E \subseteq X \times X$ of *transitions* (edges); an assignment of positive *transition probabilities* $p(x, y)$ to all transitions (x, y) so that for each state x, $\Sigma_{y \in X} p(x, y) = 1$; and an *initial probability distribution* p_0 on the states such that $\Sigma_{x \in X} p_0(x) = 1$. A *finite trajectory* of M is the finite sequence of states $\mathbf{x} = x_0, x_1, \ldots, x_n$, such that for all i, $(x_i, x_{i+1}) \in E$ and $p(x_i, x_{i+1}) > 0$. The probability of a finite trajectory $\mathbf{x} = x_0, x_1, \ldots, x_n$ is defined as $\mathbf{P}_M(\mathbf{x}) = p_0(x_0) p(x_0, x_1) \cdots p(x_{n-1}, x_n)$.

Each trajectory of C_A corresponds to an object execution. The more objects displaying the same execution behavior, the higher the probability of the associated trajectory. Hence, although the probabilities of C_A are not explicitly given, they can be learned via runtime monitoring.

Assuming that kernel-level objects have finite lifetimes (with the possible exception of objects such as the root file-system directory entry), and that state is dependent on the object's history, C_A is actually a Markov *tree*, since no object goes backward in time. The leaves of C_A fall into two categories: (i) violation-free executions of objects of type C which are deallocated after their RCs return to zero, and (ii) executions violating property stI, trI, or lkI.

Thus, a trajectory in C_A can be viewed as an object execution from its birth to its death or to an error state representing a property violation. We consider such a trajectory to be a *Bernoulli random variable* Z such that $Z = 0$ if the object terminated normally, and $Z = 1$ otherwise. Further, let p_Z be the probability that $Z = 1$ and $q_Z = p_Z - 1$ be the probability that $Z = 0$. The question then becomes: *how many random samples of Z must one take to either find a property violation or to conclude with confidence ratio δ and error margin ϵ that no such violation exists?*

To answer this question, we rely, as we did in the case of Monte Carlo model checking, on the techniques of *acceptance sampling* and *confidence interval estimation*. We first define the *geometric* random variable X, with parameter p_Z, whose value is the number of independent trials required until success, i.e., until $Z = 1$. The *probability mass function* of X is $p(N) = \mathbf{P}[X = N] = q_Z^{N-1} p_Z$, and the *cumulative distribution function* (CDF) of X is

$$F(N) = \mathbf{P}[X \leq N] = \sum_{n \leq N} p(n) = 1 - q_Z^N$$

Requiring that $F(N){=}1{-}\delta$ for *confidence ratio* δ yields:

$$N = \frac{\ln(\delta)}{\ln(1 - p_Z)}$$

which provides the number N of attempts needed to find a property violation with probability $1{-}\delta$.

In our case, p_Z is unknown. However, given *error margin* ϵ and assuming that $p_Z \geq \epsilon$, we obtain that

$$M = \frac{\ln(\delta)}{\ln(1 - \epsilon)} \geq N = \frac{\ln(\delta)}{\ln(1 - p_Z)}$$

and therefore that $\mathbf{P}[X \leq M] \geq \mathbf{P}[X \leq N] = 1 - \delta$. Summarizing, for $M = \frac{\ln(\delta)}{\ln(1-\epsilon)}$ we have:

$$p_Z \geq \epsilon \ \Rightarrow \ \mathbf{P}[X \leq M] \geq 1 - \delta \tag{1}$$

Inequality 1 gives us the minimal number of attempts M needed to achieve success with confidence ratio δ under the assumption that $p_Z \geq \epsilon$.

The standard way of discharging such an assumption is to use *statistical hypothesis testing* [12]. We define the *null hypothesis* H_0 as the assumption that $p_Z \geq \epsilon$. Rewriting inequality 1 with respect to H_0 we obtain:

$$\mathbf{P}[X \leq M \,|\, H_0] \geq 1 - \delta \tag{2}$$

We now perform M trials. If no counterexample is found, i.e., if $X > M$, then we reject H_0. This may introduce a type-I error: H_0 may be true even though we did not find a counterexample. However, the probability of making this error is bounded by δ; this is shown in inequality 3 which is obtained by taking the complement of $X \leq M$ in inequality 2:

$$\mathbf{P}[X > M \,|\, H_0] < \delta \tag{3}$$

With the above framework in place, we now present `MCM`, our Monte Carlo Monitoring algorithm. `MCM`, whose pseudo-code is given in Figure 3, utilizes DFA A to monitor properties stI, trI, and lkI, while keeping track of the number of samples taken.

```
input: $\epsilon$, $\delta$, C, t, d;
global: tn, cn;
tn = cn = ln($\delta$)/ln(1-$\epsilon$); set(timeout,d);
when (created(o:C) && flip())
  if (tn > 0) { tn--; o.to=t; o.rc=0};
when (destroyed(o:C)){
  cn--; if (cn = 0) monitoring stop;}
when (monitored(o:C) && modified(o.rc)){
  if (o'.rc < 0 || |o'.rc-o.rc| > 1) safety stop;        /* stI, trI */
  if (o.rc-o'.rc == 1) o.to = t;}
when (timeout(d))
  for each (monitored(o:C)){
  o.to--; if (o.to == 0) leak stop;}                      /* lkI */
```

Fig. 3. The `MCM` *algorithm.*

`MCM` consists of an initialization part, which sets the target (`tn`) and current (`cn`) number of samples, and a monitoring part, derived from the properties to be verified. The latter is a state machine whose transitions (`when` statements) are triggered either by actions taken by objects of type `C` or by a kernel timer thread. The timer thread wakes up every `d` time units, and the time window used to sample object executions is `t` $*$ `d`, where `t` and `d` are inputs to the algorithm. When an object `o:C` is created and the random boolean variable `flip()` is true, the target number of samples is decremented. The random variable `flip()` represents one throw of a multi-sided, unweighted coin with one labeled side, and returns true precisely when the labeled side comes up. If enough objects have been sampled (`tn=0`), no further object is monitored. For a monitored object, its reference count `rc` and timeout interval `to` are appropriately initialized. When an object is destroyed, `cn` is decremented. If the target number of samples was reached (`cn=0`), the required level of confidence is achieved and monitoring can be disabled. When the RC of a monitored object is altered, we check for a violation of safety properties stI or trI, stopping execution if one has occurred. If an object's RC is decremented, we reset its timeout interval; moreover, should its RC reach zero, the object is destroyed or reclaimed. When the timer thread awakens, we adjust the timeout interval of all monitored objects. If an object's timeout interval has expired, leak invariant lkI has been violated and the algorithm halts.

Due to the random variable `flip()`, `MCM` does not monitor every instance o of type C. Rather, it uses a *sampling-policy automaton* to determine the rate at which instances of C are sampled. For example, consider the n-state policy automaton PA_n that, in state k, $1 \leq k \leq n$, `MCM` will only sample o if `flip()`

returns true for a 2^k-sided coin. Moreover, PA_n makes a transition from state k to $k+1 \bmod n$ after exactly M samples. Hence, after M samples (without detecting an error) the algorithm uses a 4-sided coin, after $2M$ samples an 8-sided coin, etc. For a given error margin ϵ, the associated confidence ratio δ will then be $(1-\epsilon)^M$, $(1-\epsilon)^{2M}$, $(1-\epsilon)^{3M}$ and so on. PA_n also makes a transition from state k to j, where $j < k$, when an undesirable event occurs, such as a counterexample, or perhaps an execution of as yet unexecuted code. Sampling policies such as the one encoded by PA_n assure that `MCM` can adapt to environmental changes, and that the samples taken by `MCM` are mutually independent (as n tends toward infinity).

`MCM` is very efficient in both time and space. For each random sample, it suffices to store two values (old and new) of the object's RC. Moreover, the number of samples taken is bounded by M. That M is optimal follows from inequality 3, which provides a tight lower bound on the number of trials needed to achieve success with confidence ratio δ and lower bound ϵ on p_Z.

Our kernel-level implementation of `MCM` is such that if a violating trajectory is observed during monitoring, it is usually the case that a sufficient amount of diagnostic information can be gleaned from the instrumentation to pinpoint the root cause of the error. For example, if an object's RC becomes negative, the application that executed the method that led to this event can be determined.

In another example, if the object's RC fails to return to zero and a leak is suspected, diagnostic information can be attained by identifying the object's containing type. Suppose the object is an inode; we can use this information to locate the corresponding file name and link it back to the offending application.

The `MCM` algorithm of Figure 3 can be extended by expanding the class of correctness properties supported by the algorithm. The third and fourth `when` branches of the algorithm correspond to safety or bounded-liveness checks, respectively. Hence, the `MCM` algorithm can be generalized in the obvious way, to allow the treatment of arbitrary safety and bounded-liveness properties for any reactive program involving dynamic types. For example, in addition to reference counts, Aristotle currently provides Monte Carlo monitoring support for the correct manipulation of pointer variables (bounds checking), lock synchronization primitives, and memory allocation library calls. Due to its extensible, plug-in-oriented architecture, support for other properties can easily be added.

4 Implementation

In Aristotle, we instrument a program with monitoring code using a modified version of the GNU C compiler (GCC), version 4. We modified the compiler to load an arbitrary number of plug-ins and invoke code from those plug-ins at the tree-optimization phase of a compilation. At that point in the compilation, the abstract syntax tree has been translated into the GIMPLE intermediate representation [6], which includes syntactic, control-flow, and type information. A plug-in is invoked that can use the GCC APIs to inspect each function body in turn and add or remove statements. The plug-in can even invoke other GCC passes to extract information; for example, one plug-in we developed for bounds checking uses the reference-analysis pass to obtain a list of all variables used by a function.

Our use of GCC as the basis for Aristotle offers several advantages. First, it can be used to instrument any software that compiles with GCC. Prior static-checking and meta-compilation projects have used lightweight compilers [4,8] that do not support all of the language extensions and features of GCC. Many of these extensions are used by open-source software, particularly the Linux kernel. Second, the modular architecture of Aristotle allows programmers to instrument source-code without actually changing it. Third, Aristotle users can take advantage of GCC's library of optimizations and ability to generate code for many architectures. Adding GCC support for plug-ins is very simple; we added a command-line option to load a plug-in and changed the way GCC is built to expose GCC's internal APIs to plug-ins.

The information collected at the instrumented locations in the system's source code is used by runtime monitors. A runtime monitor is a static library, linked with the system at compile time. The runtime monitor contains checking code which verifies that each detected event satisfies all safety properties; furthermore, it may spawn threads that periodically verify that all bounded liveness properties hold. The monitor interfaces with the confidence engine, reporting rule violations and regulating its operation according to the confidence engine's instructions, which reflect the operation of a sampling-policy automaton. Finally, it may also perform other operations, like verbose logging and network-based error reporting, which vary from application to application.

5 Case Study: The Linux VFS

The Linux Virtual File System (VFS) is an interface layer that manages installed file systems and storage media. Its function is to provide a uniform interface to the user and to other kernel subsystems, so that data on mass storage devices can be accessed in a consistent manner. To accomplish this, the VFS maintains unified caches of information about file names and data blocks: the *dentry* and *inode* caches, respectively. The entries in these caches are shared by all file systems. The VFS and file systems use reference counts to ensure that entries are not reused without a file system's knowledge and to prioritize highly-referenced objects for retention in main memory as opposed to being swapped out.

The fact that these caches are shared by different file systems, implemented by different authors and of varying degrees of maturity, introduces the potential for system resource leaks and faults arising from misuse of cached objects.

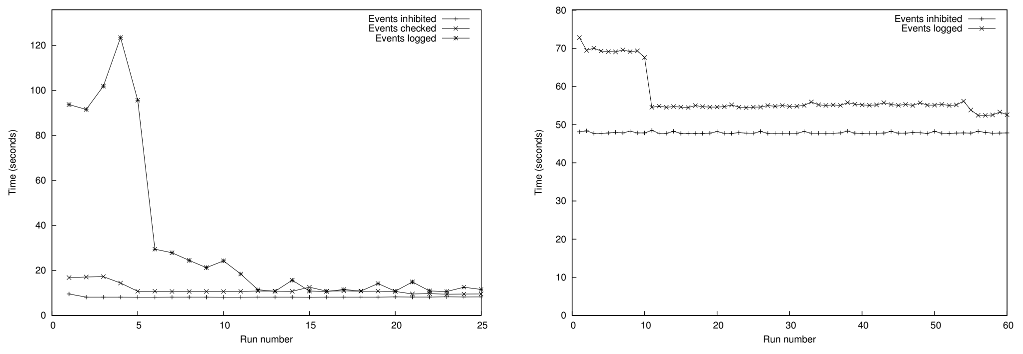
For example, a misbehaving file system may prevent a storage device from being safely removed because the reference count for an object stored to that device was not safely reduced to zero. Worse, a misbehaving file system could hamper the performance of other file systems by failing to decrement the reference counts of cache data structures.

Using the Aristotle framework, we developed a tool that monitors reference counts in the Linux VFS. As described in Section 3, we enforced a state invariant (stI), a transition invariant (trI), and a leak invariant (lkI).

The plug-in for this case study instruments every point in the source code at which a reference count was modified. Because we had access to type information, we were able to classify reference counts for dentry and inode objects. Whenever it is invoked, the runtime monitor checks the operation to ensure that the safety properties hold. Additionally, if the operation is a decrement, the monitor updates a timestamp for that reference count, which is maintained in an auxiliary data structure. A separate thread periodically traverses the data structure to verify that all reference counts have been decremented more recently than time interval T. Additionally, all checked operations are optionally logged to disk.

The confidence engine maintains separate confidence levels for dentry and inode reference counts using our Monte Carlo model checking algorithm. For clarity, we demonstrate the system with a sampling policy automaton that disables checking when a 99.999% confidence level has been reached that the error rate for that reference counter category is less than 1 in 10^5 samples. As discussed in Section 3, a sample is defined as the lifetime of a cached object, that is, the period when the object's reference counter is nonzero. Other sampling policies, such as flipping an n-sided coin where n increases as confidence increases to determine whether to sample a given object, allow more fine-grained trade-offs of performance vs. confidence; additionally, it may be advisable to increase the sampling rate as the environment changes.

Figure 4(a) shows the performance overhead of the system with logging and checking enabled, logging disabled but checking enabled, and no instrumentation, under a micro-benchmark designed to exercise the file system caches. In each run, the micro-benchmark creates a tree of directories, does a depth-first traversal of that tree, and deletes the tree. Because directories are being created and deleted, on-disk data is being manipulated, causing creation and deletion of objects in the inode cache. Additionally, the directory traversal stress-tests the dentry cache. We observe an initial 10x overhead as both dentry and inode reference counts are being monitored and all accesses are being logged. After five runs, which take six minutes in total, dentry confidence reaches the target, and overhead falls to a factor of three. Finally, five minutes later, after eleven runs, overhead drops to 33% when inode confidence reaches the target. The remaining overhead is a characteristic of our prototype; we expect optimization to reduce it significantly.



(a) directory-tree microbenchmark (b) compilation of GNU `tar`

Fig. 4. Overhead reduction as confidence increases.

Figure 4(b) shows the effects under a benchmark that puts less stress on the file system. Compiling the GNU `tar` utility involves less cache activity than the micro-benchmark described above, so the overheads from monitoring are lower; however, it also takes longer for confidence to reach the target. Initial overhead with logging was 46%. After ten runs, or eleven minutes, this overhead dropped to 14% as dentry confidence reached the target. Forty minutes later, at the 55th run, overheads dropped to 11% as inode confidence reached its target as well.

6 Related Work

Runtime verification is the subject of much recent research [3,14,15]. Our work combines and takes these concepts one step further, detecting instrumentation points at compile time and managing itself autonomously at runtime using statistically-driven sampling policies. Other related work includes metacompilation [8] and ESP [11], which extend compilers with static checkers to find violations of system-specific properties; and the model-checking efforts directed at network protocols [5,13], file systems [17], and device drivers [1].

Chilimbi and Hauswirth [9] have implemented a sampling-based technique for detecting memory leaks in programs. They maintain a timestamp with each memory object and a sampling rate with each basic block of code. Each time a basic block b makes a reference to an object o, o's timestamp is updated and b's sampling rate is decreased. No attempt is made to quantify the confidence level and error margin introduced by this technique.

7 Conclusions

We have presented the `MCM` algorithm for Monte Carlo monitoring and runtime verification, which uses sampling-policy automata to vary its sampling rate dynamically as a function of the current confidence in the monitored system's correctness. We implemented `MCM` within the Aristotle tool environment, an extensible, GCC-based architecture for instrumenting C programs for the purposes of runtime monitoring. Aristotle realizes this architecture via a simple modification of GCC that allows one to load an arbitrary number of plug-ins dynamically and invoke code from those plug-ins at the tree-optimization phase of compilation. Our experimental results show that Aristotle reduces the runtime overhead due to monitoring, which is initially high when confidence is low, to long-term acceptable levels as confidence in the deployed system grows.

As future work, we are developing an instrumentation-specification language to facilitate plug-in construction and insertion into GCC. Additionally, we are investigating the integration of auxiliary information, such as code coverage, into sampling policies. This would allow, for example, instrumentation to be increased when a rarely-used section of code is executed.

8 Acknowledgments

Yanhong A. Liu and Scott D. Stoller provided valuable feedback to the architectural model described in Section 2 and are collaborating with us on the future work described in Section 7. We are also grateful to the anonymous reviewers who provided invaluable feedback that helped us present our work in as clear a manner as possible.

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

References

[1] Ball, T. and S. K. Rajamani, *The SLAM toolkit*, in: *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification* (2001), pp. 260–264.

[2] Bernstein, A. and J. P. K. Harter, *Proving real-time properties of programs with temporal logic*, in: *SOSP '81: Proceedings of the eighth ACM symposium on Operating systems principles* (1981), pp. 1–11.

[3] Bodden, E., *A lightweight LTL runtime verification tool for Java*, in: *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (2004), pp. 306–307.

[4] C. W. Fraser and D. R. Hanson, "A Retargetable C Compiler: Design and Implementation," Addison-Wesley Longman Publishing Co., Inc., 1995.

[5] Dong, Y., X. Du, G. Holzmann and S. A. Smolka, *Fighting Livelock in the i-Protocol: A Case Study in Explicit-State Model Checking*, Software Tools for Technology Transfer **4** (2003).

[6] GCC team, T., "GCC online documentation," (2005), http://gcc.gnu.org/onlinedocs/.

[7] Grosu, R. and S. A. Smolka, *Monte carlo model checking (extended version)*, in: LNCS 3440 on SpringerLink (2004), pp. 271–286.

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

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

[10] Kupferman, O. and M. Y. Vardi, *Model Checking of Safety Properties*, Formal Methods in System Design **19** (2001), pp. 291–314.

[11] M. Das and S. Lerner and M. Seigle, *ESP: Path-Sensitive Program Verification in Polynomial Time*, in: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2002, pp. 57–68.

[12] Mood, A. M., F. Graybill and D. Boes, "Introduction to the Theory of Statistics," McGraw-Hill Series in Probability and Statistics, 1974.

[13] Musuvathi, M., D. Y. W. Park, A. Chou, D. R. Engler and D. L. Dill, *CMC: A Pragmatic Approach to Model Checking Real Code*, in: *Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI '02)* (2002), pp. 75–88.

[14] Rosu, G. and K. Sen, *An Instrumentation Technique for Online Analysis of Multithreaded Programs*, in: *18th International Parallel and Distributed Processing Symposium*, 2004, p. 268b.

[15] Sammapun, U., A. Easwaran, I. Lee and O. Sokolsky, *Simulation of Simultaneous Events in Regular Expressions for Run-Time Verification*, in: *Proceeding of Runtime Verification Workshop (RV'04)*, Barcelona, Spain, 2004, pp. 123–143.

[16] Vardi, M. Y. and P. Wolper, *An Automata-Theoretic Approach to Automatic Program Verification*, in: *Proceedings of the Symposium on Logic in Computer Science (LICS)*, Cambridge, MA, 1986, pp. 332–344.

[17] Yang, J., P. Twohey, D. R. Engler and M. Musuvathi, *Using Model Checking to Find Serious File System Errors*, in: *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 2004)*, San Francisco, CA, 2004, pp. 273–288.