Remote Debugging with Controllable Overhead

A Thesis Proposal Presented

By

Sean Callanan

Technical Report FSL-08-02

Mar 7, 2008

Abstract

Remote devices, a category that includes smart-phones and consumer-level wireless routers as well as servers in a datacenter and consumer PCs, are becoming an increasingly relevant target for debugging. In particular, with operating systems like Mac OS X, Linux, and Windows Mobile being used in the most recent generation of smart-phones, the OS platform for smart-phones has become essentially the same as that for the workstations used to develop their software. Also, consumer wireless routers are increasingly using Linux as the basis of their software stack. This suggests that it should be possible to create a debugging model for errors that appear in deployment.

However, two obstacles make debugging these devices interactively a challenge. First, for consumer devices the user interface itself frequently lacks even the most rudimentary debugging support, such as a shell prompt. As an aggregate, servers in a datacenter have remote management infrastructure, but this is only designed for high-level status monitoring. Second, even if the user interface includes such support, the user lacks the expertise to use a debugger effectively. As a result of these two factors alone, an interactive debugger is ineffective except in a development environment. Some form of asynchronous debugging — that is, inserting debugging code and logging its results — is necessary.

In this thesis proposal, we confront the problems with asynchronous debugging in these contexts. Specifically, we address the following issues: completeness, efficiency, and versatility. Completeness means having access to as much information as possible about the execution of the program being debugged. We solve the completeness problem by modifying the GNU compiler to produce instrumented versions of programs, exposing full information about every control-flow and data-flow event. Efficiency means doing this with predictable impact on runtime performance and code size — a particular concern on devices with constrained CPU, network, and storage resources. We solve the efficiency problem by guaranteeing constant overhead and only keeping instrumented versions of functions that are currently being debugged on the device. Versatility means allowing the developer of the software to control what is being debugged. We solve the versatility problem by providing a central server or distributed server infrastructure that records the debugging requirements for the code and interacts with devices in the field, providing them with necessary code and handling the logs they produce.

We present our existing work on efficient instrumentation, and then go on to propose a system that addresses the above issues in a coherent, extensible manner, making it possible to debug even widely-distributed devices with ease. The solutions we design will be applicable not only to simple networked devices like cellular phones and wireless routers, but to any computer systems whose users are incapable of debugging the software on their own.

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

Contents

List of Figures

Chapter 1

Introduction

In this thesis, we are proposing a new model for conducting cooperative debugging. Cooperative debugging is a recent development in software debugging, and gives an application's developers the ability to investigate problems in that program's execution on many computers [29]. In cooperative debugging, developers deliver versions of their programs that they have modified to record information about their execution. Periodically, these programs report that information back to the developers, who then use it to draw conclusions about the execution of the program.

Cooperative debugging is useful for a variety of technical reasons; we will discuss three of these in detail, and also mention related work that tries to solve similar problems. We summarize these advantages as follows:

- Cooperative debugging allows developers to examine the execution of their applications in environments that they cannot replicate in-house. For example, the space of possible configurations a video player may encounter is staggering: container formats, video and audio encoding schemes, bit-rates, storage devices, and computer configurations can all vary.
- Cooperative debugging takes the user out of the diagnosis process. Most bug report forms rely on the user to aid in reproducing the bug by providing extensive configuration information. Even if the user takes the time to fill out such a form and provides all relevant information, a developer may want to get more specific information. Unless the bug is preventing the user from doing something—in which case the user will cooperate until the issue is resolved—it is difficult to convince the user to take time to help in diagnosing it.
- Cooperative debugging permits debugging on a scale that is impossible for all but the largest organizations. Even if configurations are exactly identical, deadlocks and race conditions may occur only extremely rarely, and developers will find it difficult to identify them without a large database of program executions to refer to. In addition, large databases of executions can provide compelling justification for rearchitecting large subsystems of a program.

If we translate these advantages into economic terms, cooperative debugging reduces the amount of resources required to achieve a given level of value from debugging by enlisting users as testers. Furthermore, it allows on-staff QA professionals and programmers to spend their time more effectively by indicating which bugs are encountered most often in the field.

In Figure 1.1, we present a model of flow of information through a cooperative debugging system. We will use the various components of this model to discuss past solutions to this problem.

In cooperative debugging, the developer prepares an application's source code normally. The

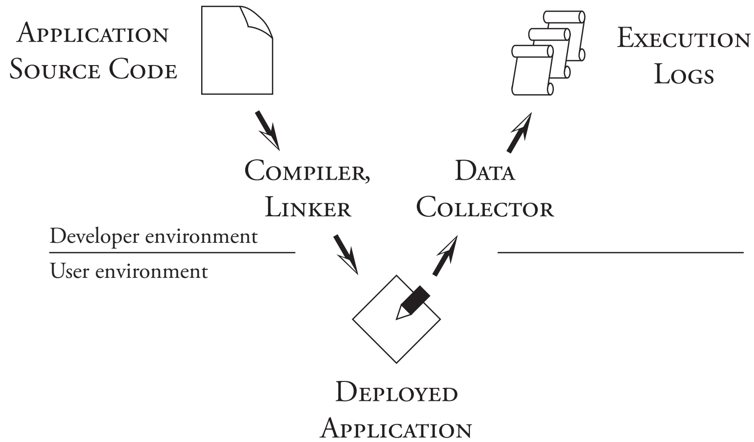


Figure 1.1: A high-level view of a cooperative debugging system.

compiler and *linker*, in addition to performing their normal tasks of translating the program into machine code and preparing it to run in the target execution environment, prepare the application for cooperative debugging by adding instrumentation code that executes at specific points in the application's execution and collects information that the developer finds relevant. The application, once shipped to consumers, compiles logs of its execution that it sends back to the developer.

Chapter 2

Instrumentation and Debugging

We discuss existing approaches to compilation in Section 2.1 and debugging on the target machine in Section 2.2.

2.1 Compiler-based Instrumentation

All modern compilers have special support for debugging. The reason for this is that the compiler is typically the first part of the development "toolchain" to analyze an application, and maps programmer-generated artifacts to executable code. In order to fix problems with the executable code, developers must determine the relationship between the portion of the executable that failed and the artifacts that it was generated from. Consequently, most compiler store a mapping from artifacts to executable code; this mapping is typically known as *debugging information*.

Compilers typically insert debugging information into data packets that reside alongside the code they describe [37], or into separate sections of a binary file [21]. This debugging information documents several aspects of the source-executable mapping:

Line information: The compiler records which ranges of lines in the assembly code correspond to particular lines of the original source code. The DWARF format [21] also includes column information, to identify assembly instructions that correspond to individual portions of complex explanations, such as the individual clauses in a `for` statement.

Variable information: The compiler saves information about the local variables for a function, as well as the location of static and global variables. This information includes whether the variables are allocated on the stack or in registers, how they can be extracted, and what type they are. For variables that move between the stack and registers, DWARF allows compilers to emit *location lists*.

Function descriptions: To facilitate calling of functions and stack unwinding (see Section 2.2, compilers can emit function signatures that specify how functions should be called and where their code resides.

The usage model for debugging information is very specific: debuggers use it to inspect and manipulate program's state when it is paused. This inspection and manipulation is either programmer-guided or very naïve, as we shall see in Section 2.2. For more sophisticated analyses, particularly those that occur without pausing the software, the compiler needs to not only add auxiliary information but also modify the application so that it performs these analyses—or provides the data required to perform them—at run time.

In order to understand how the compiler does this, we must first explain the *intermediate representations* that the compiler maintains for an application as it transforms it into executable code. An intermediate representation is a data format (typically memory-resident) that serves as the interface between two parts of the compiler, or to permit a user-specifiable combination of similar components, such as optimizers, to operate on a portion of the software sequentially.

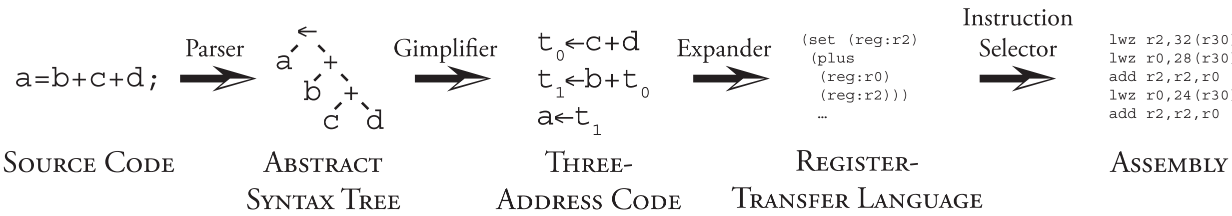


Figure 2.1: The architecture of GCC, with the intermediate representations it uses.

Our compiler-based instrumentation work has focused on the GNU Compiler Collection (GCC), whose components are illustrated in Figure 2.1. We now discuss the individual intermediate representations, and describe their respective characteristics and how each of them can be used for debugging.

Abstract Syntax Trees: After parsing, GCC (like most compilers) represents a source file as a collection of trees that represent the syntactic structure of each function. These trees retain much of the original structure of the source file, including block information, loop structure and nesting, and compound statements. At the abstract syntax tree layer, transformations can most easily detect common programming idioms—such as use of iterators in a loop—without performing expensive and complicated analyses. Instrumentation, like source-level tracing, that reports information at the granularity of programmer-written code lines is best implemented at this level. Other tools, such as Cil [16], represent code in a similar way.

Three-Address Code: GCC converts the abstract-syntax trees into their semantic equivalents, reducing them to assignments that take at most two values per statement and combines them to produce a third. The assignments retain all the type information that was discovered during parsing, and share the building blocks of the abstract-syntax tree representation, but are much simpler to manipulate. To represent intermediate values, GCC generates temporary variables; additionally, it simplifies the control flow structure by constructing a *control-flow graph* and replacing more sophisticated structures with conditional `goto`s. For optimizations, GCC developers recommend this layer, which was designed explicitly for ease of manipulation by the programmer [35]. The reason for this is that most data-flow and control-flow analyses use a control-flow graph representation and prefer as simple a syntactic structure as possible because this reduces the number of side-effects that must be considered.

Register-Transfer Language: After giving optimizers the opportunity to transform the three-address code, GCC converts it into a format that closely resembles assembly code. It determines the kind of storage that each variable requires, what low-level operations must be performed on the variables, what classes of instructions provide them, and performs register allocation and final instruction selection based on this information. Although GCC performs some optimizations at this level, including instruction scheduling, the implementation of RTL in GCC is so complicated and fragile that GCC programmers recommend against using it for any kind of instrumentation.

We developed a plug-in based transformation system based on the GCC's GIMPLE intermediate representation [33], which it uses for three-address code. Despite its simplicity, the GIMPLE intermediate representation presents its own challenges during transformation development and testing for several reasons. First, the maturity of the GCC project and the fact that many system distributions depend on GCC to compile their system makes it difficult to get transformations integrated into GCC until they are very mature. Second, it may not be desirable to include and maintain transformations that do not have broad appeal in the core GCC distribution. Finally, it is an unattractive proposition to have to distribute experimental transformations as patches against a particular version of GCC and recompile the compiler when changes are made.

To solve these problems, we developed a plug-in system similar to that used by Eclipse [30]. Our system allows separate development and compilation of GIMPLE transformations, solving the problems listed above and offering new features like enhanced debuggability and better argument passing. We have already developed a variety of plug-ins using our system, and have realized two main benefits. First, we were able to take advantage of graphical debugging tools that we describe in Section 2.1.2 as well as significantly reduced development time because we were developing outside the GCC build system. Second, we were able to port our transformations from one version of GCC to another without changing a *single* line of code; once the plug-in support was ported to the new GCC release, the plug-ins just needed to be recompiled.

In the remainder of this section, we demonstrate the simplicity and power of GCC transformation plug-ins. In Section 2.1.1, we describe the modifications to GCC that make plug-in–based development possible. In Section 2.1.2, we describe some plug-ins that we have already built using this infrastructure, highlighting plug-ins that are useful to transformation developers. In Section 2.1.3, we discuss two parts of GCC that could be made into plug-ins. In Section 2.1.4, we describe plug-ins that could be created in the future, and we conclude in Section 2.1.5.

2.1.1 Modifications to GCC

Plug-ins are built based on an Autoconf-based template [11]. The template's `configure` script currently requires the headers from a built version of the GCC source code; when the plug-in is built, the Makefiles produce a shared object file suitable for loading using the host operating system's dynamic loader interface.

Only minor changes need to be made to GCC to support plug-in loading. These changes revolve around three tasks; we will discuss them below in turn. The first change is an addition to the GCC build sequence, compiling the Libtool `ltdl` library [14] into GCC and linking GCC with `-export-dynamic`. This allows GCC to load plug-ins, and allows plug-ins to access GCC interfaces. The second change is the addition of an optimization pass before all other GIMPLE transformations, and at the start and end of translation for each file. This allows plug-ins to maintain per-file state and perform code optimizations while referring to this state. The third change is the addition of a compiler flag that allows the user to specify plug-ins to load and provide arguments to those plug-ins either on the command line or through files.

To add the `ltdl` library to GCC, we modified the top-level top-level Makefile to add build rules for the `ltdl` library. Additionally, we modified the build rules for the `cc1` binary to make it compile with Libtool, export its symbols like a shared library (using the `-export-dynamic` option to Libtool), and use the `ltdl` library to load plug-ins. The ability to export symbols from an executable to plug-ins does not exist on every platform: Linux, Solaris, and Mac OS X support this functionality, for instance, but Cygwin does not. A build process in which the GCC backend code is linked as a shared library, and `cc1` and all plug-ins are linked against it, would have eliminated this requirement. However, large amounts of state that is currently maintained as globals by the backend would have to be converted to on-stack state because otherwise `cc1` and the plug-in would have differing copies of the backend's global state.

To allow instrumentation plug-ins to run at the proper times, we added several new passes to `passes.c`, allowing plug-ins to run at various points in compilation. We describe these in Table 2.2.

Pass	Location	Purpose
`pre`	Before compilation	Allows plug-ins to perform data-structure initialization before compilation of a source file begins.
`ctrees`	After C parsing	Allows plug-ins to analysze the abstract syntax trees for a C program before it is converted to GIMPLE.
`cgraph`	Before IPA transformations	Allows plug-ins to perform *interprocedural analyses*, which have access to every function's code and the call graph.
`gimple`	After IPA, per function	Allows plug-ins to manipulate the GIMPLE representation of each function, including the control-flow graph.
`rtl`	After conversion to RTL, per function	Allows plug-ins to manipulate the RTL for each function.
`post`	After compilation	Allows plug-ins to perform data-structure cleanup after compilation of a source file ends successfully.

Figure 2.2: Plug-in transformation passes.

Finally, to allow the end user to specify which plug-ins should be loaded with which arguments, we provided a new argument, `-ftree-plugin`, which has the syntax shown in Figure 2.3.

```
-ftree-plugin=plug-in-name
    :key=value
    :...
```

Figure 2.3: Syntax for specifying a plug-in.

The first argument, *plug-in-name*, is a shared object file that contains functions for one or more of the passes described in Table 2.2. The list of *key-value* pairs specifies arguments to the plug-in; these can be fetched using a function. In addition, the special key `_CONF` specifies a file to be loaded and parsed for additional arguments; in this case, each line in the file is a key-value pair separated by an = sign.

2.1.2 Existing plug-ins

We will now enumerate some plug-ins that we have already developed: a verbose dump plug-in for GIMPLE meant for use by programmers in developing transformations, and a call-trace plug-in for use by end users in tracing their code. We have also developed `malloc` checking and bounds-checking plug-ins; however, these will be superseded by a plug-in implementation of Mudflap (see Section 2.1.4).

Verbose Dump Plug-in. Transformation developers frequently require a view of the GIMPLE code that is as verbose as possible. They use this view for several purposes: to identify patterns that need to be transformed, to determine the proper form of GIMPLE structures that transformations should generate, and to verify that transformations are working correctly. We designed a verbose dump plug-in to facilitate this. We designed the verbose dump plug-in with extensibility in mind: as GIMPLE evolves and grows, the verbose dump plug-in will handle new GIMPLE objects, such as new tree codes or parameters, with little or no changes needing to be made. We achieved this by creating a new file, `parameter.def`, that resembles `tree.def` but formally specifies all the accessor macros that exist for tree attributes. The file contains lines of the form shown in Figure 2.4.

```
DEFTREEPARAMETER(
   name,
   type,
   macro,
   code, ...
)
```

Figure 2.4: Syntax of `parameter.def`

The *name* field specifies the name of the macro; the *type* field specifies what type of data it returns (e.g., `SIZE_T` or `TREE`); the *macro* field specifies the macro used to extract the field; and the *code* fields constitute a list of `TREE_CODE`s for trees that have this parameter. For example, the parameter named `type_precision` has type `SIZE_T`, macro `TYPE_PRECISION`, and codes `INTEGER_TYPE`, `REAL_TYPE`, and `VECTOR_TYPE`.

Graphical Inspection of GIMPLE Code. As shown in Figure 2.5, the output from the verbose-dump plug-in is so verbose as to be overwhelming in large quantities. Rather than adopt a simplified representation, we instead developed a Java-based tool called Gimple Viz to represent the output graphically. We chose Java as the development language due to its cross-platform compatibility, which allowed us to concentrate on the development of the actual tool itself as opposed to platform support and library dependencies. Figure 2.6 is a screen-shot of Gimple Viz displaying a file. The visualizer has three main areas: the Control Flow Graph area, the GIMPLE Tree View area, and the Source / Search area, which we describe below.

The control flow graph for each function is rendered as rectangles connected by arrows. Each colored rectangle represents a basic block. When the user clicks on a block, Gimple Viz highlights the selected block along with its predecessors and successors. The successor edges are highlighted as well. Additionally, it displays a tree representation of the corresponding GIMPLE nodes in the GIMPLE tree view area, and highlights corresponding code or dump lines in the source / search area.

The GIMPLE tree view area is a visual representation of the GIMPLE code for a particular basic block. The root node of each tree is a statement from the currently selected basic block, labeled with the result of applying `print_generic_stmt`. The other nodes are operands or parameters of their parents. The user interacts with the tree view in two ways: clicking and searching. Manually clicking a node will expand that node showing its children. This process can be repeated until the desired node is reached. Searching for a particular `TREE_CODE` will expand the tree to reveal the desired node, allowing the user to quickly locate specific nodes.

The Source/Search area can show search results, source code, and verbose-dump output. The results of searches—function searches, basic-block searches, and type searches—are interactive: clicking on a function search result shows the control-flow graph for that function; clicking on a basic-block search result shows the containing function's CFG and highlights the block; and

```
MODIFY_EXPR 1,2
 TREE_TYPE:
   INTEGER_TYPE 2,0
    TYPE_PRECISION=32
   TYPE_UNSIGNED=true
 VAR_DECL 2,0
  TREE_TYPE:
   INTEGER_TYPE 2,0
    TYPE_PRECISION=32
    TYPE_UNSIGNED=true
  DECL_ARTIFICIAL=true
 MULT_EXPR 1,2
  TREE_TYPE:
   INTEGER_TYPE 2,0
    TYPE_PRECISION=32
    TYPE_UNSIGNED=true
```

Figure 2.5: A portion of the verbose dump output for one statement, leaving many node attributes out.

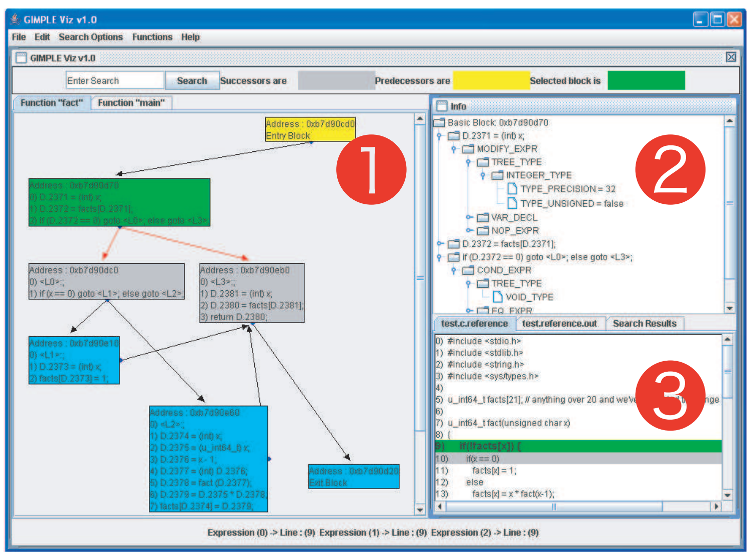


Figure 2.6: Gimple Viz displaying a file. 1 marks the CFG area, 2 marks the tree view, and 3 marks the source/search area.

clicking on a TREE_CODE search highlights the containing basic block and expands the containing tree in the GIMPLE tree view to make the tree with that code visible.

Gimple Viz can also display the original source file that was compiled by GCC in the source / search window. For quick reference, line numbers are displayed for the user. Although the user cannot directly interact with this area, clicking a basic block or a search result will highlight the lines corresponding to that block, its predecessors and its successors. Finally, the source / search window can also display the raw verbose dump output.

Call Trace Plug-in. We have developed a plug-in called *call-trace* to allow full verbose tracing statements to be added to a program at compile time without requiring the programmer to add

```
*** CALL TO main [0]
Struct :**test** found in function
**main**
test->value = (int32_t)5
test->name = (char*)"contents"
** CALL TO foo [1]
* [1] testPtr = 0x0x7fffcef31770
*** CALL TO foo2 [2]
* [2] value = 0x0x7fffcef31748
Conditional found value = FALSE,
right branch taken...
*** [2] RETURNED null
```

Figure 2.7: Call trace output

any code. This feature significantly reduces debugging time for many code problems by eliminating the need to add `printf` statements and other debugging statements to code, and by providing verbose tracing information in cases where the programmer would normally have needed to single-step the program in `gdb`.

This plug-in identifies control points in the GIMPLE code corresponding to conditional statements and function calls, as well as accesses to variables. Arguments control exactly which statements are logged, and which portions of the source code are to have logging added. The way events are reported is also configurable: logging statements can be printed using `fprintf` or sent to a custom logging function. Figure 2.7 shows sample output from the call tracer.

We are currently developing an extension to Gimple Viz to display the output from the call-trace plug-in in a visual manner, giving the developer the ability to watch the internal execution of a program at run-time. We are also expanding the call-trace plug-in to detect not only conditionals but loops as well by tying into the GENERIC intermediate representation.

2.1.3 Making Plug-ins from Existing Functionality

In this section, we describe portions of GCC's functionality that could be extracted into separate modules for use only when needed. This would have three benefits: first, it would enforce modularity for these components, ensuring that they can be maintained separately from the main code base and contributing to their stability as GCC internals change. Second, it would reduce the turnaround time for fixes to mudflap because they would not need to be subject to the scrutiny that core GCC patches are subjected to. Third, it would reduce the size of the core GCC code base, resulting in less code for GCC's core developers to maintain and support, and less download and compilation time for end-users.

Mudflap. This utility provides pointer-debugging functionality including buffer overflow detection, matching-based leak detection, and reads to uninitialized objects. It is implemented as two GIMPLE optimization passes: one that executes before the lowering to SSA (Static Single Assignment) so that all scopes are intact, and one that executes after lowering and optimization to get an accurate view of just those memory accesses that have actually been performed. Mudflap can be converted to a plug-in provided that plug-in hooks are provided at multiple stages in the optimization process. Our plug-in infrastructure supports transformation hooks at all locations where built-in GIMPLE transformations can take place, making this process straightforward.

gcov and gprof. These utilities consume call-graph information that is generated by GCC and by the running program, creating runtime profiles of the execution patterns for code that has been compiled with the `-p` or `-fprofile-arcs` flags. When profiling, GCC modifies the program to include *coverage counters* embedded in the program that provide runtime coverage information. It also generates a call-graph for the program. The transformation that performs these tasks runs as a transformation in a way analogous to Mudflap, but labels basic block edges with additional information that uses the `aux` field in the basic block structure. This does not present a problem for these transformations, since they take place in one pass and do not need persistent `aux` storage. However, other plug-ins that may need to do analyses at multiple times in compilation it may become desirable to expand `aux` to support addition of custom fields, perhaps keyed on a string, at runtime.

2.1.4 Future Work

Once the groundwork is in place that allows GCC transformations to be developed as plug-ins, we anticipate that many new transformations will be developed. In this section, we outline future applications of plug-ins, some of which we are currently developing for our own research.

Transformations in Python. Some developers only want to perform straightforward analyses or transformations that use the GIMPLE API. To reduce development time for these developers, we are developing a plug-in that will expose the GIMPLE API to Python scripts. This plug-in links against the Python library and executes a user-specified Python script for each function being translated. It currently allows read-only access to basic blocks and trees; we are adding support for viewing and editing the control-flow graph, adding and removing statements, and modifying trees. In addition to reducing development time, this plug-in will allow developers to use Python data structures, reducing implementation time for optimizations that use sophisticated algorithms to perform static analyses on GIMPLE.

Library call error detection. When developing systems software, programmers frequently add large amounts of error checking for library function calls to detect problems that are ironed out in the early stages of development. This error-checking adds to code size, reduces code readability, and takes time. In addition, retroactively adding error-checking onto existing code if it fails can be a significant time investment. . A GIMPLE transformation plug-in could be used to add error-checking to code at compile time, optionally warning when the code is not written to check the result of calls that commonly fail.

Interface profiling. Threaded applications typically have points at which threads wait for responses from other threads. These can take several forms: functions that are called to perform synchronous requests, or locks that the programs block on until data is ready. Additionally, even single-process applications can spend time waiting for library functions or system calls to complete. A GIMPLE transformation plug-in could accept a list of locks and interface functions to profile, and add entry-exit profiling to these locks and functions. This would be coupled to a runtime library that determines the amount of time spent waiting for these interfaces, credited to the functions that waited for them.

2.1.5 Conclusion

We have described a framework that we developed that allows GCC to load and execute plug-ins that implement custom GIMPLE transformations. This framework offers three compelling benefits:

- it reduces development time for new GCC transformations;

- it allows transformations to be developed and distributed that would otherwise be difficult to use or not available at all; and
- it reduces the workload for the GCC core developers by reducing GCC code size and allowing many transformations to be maintained separately.

We have shown a verbose-dump plug-in and a compatible Java-based visualizer that help GCC developers develop and debug their transformations. We have also shown a call-trace plug-in that tracks function calls, variable accesses, and conditionals, providing a detailed view of the execution of a program. In addition to these existing plug-ins, we have shown examples of existing functionality in GCC that could be converted to plug-ins, and examples of new functionality that do not exist yet but would be well-suited to implementation as plug-ins.

2.2 Debugging

Traditionally, debuggers have been defined as command-line tools that have a relatively standard set of functions, controlled by a command-line interpreter. This kind of debugger is also called an *source-level debugger*. The interpreter evaluates expressions written in a language very similar to the original source code, using type information embedded by the compiler and data from a paused process or process image. Examples of source-level debuggers include GDB [15] and dbx [45]. The core mechansims used by these tools are:

Process control: Modern UNIX kernels provide debuggers with interfaces to stop, start, and single-step processes. Most other operations a debugger performs require a process to be paused first. Operating-system support for process control varies widely. Linux provides a system call, `ptrace`, which implements process stopping, starting, and single-stepping as kernel facilities [22]. Solaris exposes this functionality to user-space through the `/proc/`*pid*`/ctl` file, which is replicated for every process and every thread [31]. Mac OS X exports this functionality through the Mach task and thread port interfaces [2].

Watchpoints and signal handling: Users frequently want to stop a process when a particular event occurs, including when a signal arrives and when the process touches a particular area of memory. Debuggers can trace signals in various ways: Linux and Mac OS X provide the ability to trace signals through the `ptrace` interface. (Mac OS X does not provide Mach interfaces to do so because Mach does not use signals.) Solaris provides signal tracing via the `/proc/`*pid*`/ctl` interface. Watchpoints are typically implemented using dedicated CPU support, typically implemented as watch registers [51].

Memory inspection: Reading and writing a process's memory violates isolation, so this too requires special kernel support. It is nonetheless necessary for inspection of a program's variables, as well as the debugging information necessary to interpret the program's execution state. Linux and Solaris provide a special file that allows a debugger to read from a process's memory (`/proc/`*pid*`/mem` on Linux, and `/proc/`*pid*`/as` on Solaris), although Linux's version does not allow writing due to a security hole. Mac OS X provides a Mach call, `mach_vm_remap`, to map another task's memory into the current task's address space [42].

Source-level debuggers are a well-understood area; however, there are three areas in which they fall short of the requirements of programmers today. First, the source-level debugger interface is locked into a particular interaction model. Modern development environments such as Xcode [25] and Eclipse [17] must consequently rely on slow, text-only interfaces to the debugger. Second, source-level debugging is slow; the fact that the debugger is external to the process and typically interprets its command language makes it unsuitable for operations that need to be performed often, like verifying that a particular lock is held each time a variable is accessed [43]. Third, source-level debugging does not permit maintenance of auxiliary data structures to keep track of information, preventing checks like stale memory detection.

In this section, we will discuss existing alternatives to source-level debugging that others have developed, as well as interposition methods we have developed.

2.2.1 Operating system and hardware-assisted debugging

As we saw in our discussion of source-level debuggers, operating systems and hardware provide considerable support for debugging, which is exploited by a variety of tools; we will discuss some of these below.

The Solaris modular debugger. Solaris provides extensive user-level support for debugging [32]. The Solaris modular debugger, `mdb`, provides an API that allows programmers to write custom debugging tools that exploit these APIs [41]. Programmers implement these tools as *modules*, which they can compile as object files using a C compiler and load into `mdb`, allowing maintenance of auxiliary data structures using the standard C heap allocation APIs. The debugger provides a uniform command-line interface that allows users to compose the functionality provided by these modules, allowing quick inspection of large data structures and maintenance of auxiliary data structures. `mdb` can debug both userspace targets and the Solaris kernel, and Solaris developers have provided an extensive module infrastructure for debugging memory, inter-process communications, and other subsystems.

DTrace. Although its compiled module support makes `mdb` much more versatile than conventional source-level debuggers, `mdb` still pauses the process to perform its inspection. This makes `mdb`'s approach unsuitable for production environments and for diagnosing timing-sensitive bugs like performance bottlenecks and deadlocks. DTrace is an event-processing system that is intended to operate autonomously on a running system [9]. It runs inside the operating system's kernel, and can monitor events both in the kernel and in user processes. Users interact with DTrace by writing scripts and compiling them into a restricted bytecode, which runs in a virtual machine inside the kernel, reducing context-switch latencies and performing data collection and aggregation without requiring interaction with the user.

Hardware counter overflow profiling. Many microprocessors include performance counters that record instruction counts, cache misses, cache invalidations, pipeline stalls, branch mispredictions, and other statistics [28]. These counters increment each time a particular event occurs, and often generate CPU interrupts when they overflow. The operating system can read from and write to these registers. Many tools use these counters to measure overall execution characteristics [10], allowing the developer to get a general view of an application's execution characteristics. In cases where a counter overflow generates an interrupt, many tools also provide the ability to sample these events to determine their causes, by loading the counter with a value very close to an overflow and inspecting the code that caused the interrupt when an overflow occurs [54].

Hardware-assisted memory profiling. All modern microprocessors intended for server or desktop use have memory-management units. Operating systems take advantage of this hardware not only to provide isolation, but to measure applications' usage of memory. The canonical two-handed clock algorithm takes advantage of MMU page usage bits, swapping out unused pages [47]. Although the granularity of this approach is limited to the system's page size, SafeMem demonstrates that similar protections are possible using ECC memory, providing cache-line granularity [38]. To determine whether a particular cache-line is used, SafeMem disables ECC on it, scrambles it in a reversible manner, and then re-enables ECC. The ECC check is performed on the cache line when it is used; cache lines for which no ECC errors come in are consequently not being used. SafeMem uses this approach for bounds-checking as well, putting bad data into ECC lines to either side of a valid allocation. ElectricFence applies a similar approach using only the MMU [36].

2.2.2 Binary modification

When specific hardware support is unavailable, or to avoid the kernel-user context switches associated with hardware interaction, debuggers can rewrite the binary representation of the program being debugged. This can be accomplished in one of two ways. First, a tool can modify a binary before use; second, a tool can instrument a running binary. We will discuss examples of both.

ATOM. ATOM is a library that programmers can use to implement instrumentation tools [44]. Tools built with ATOM accept a program binary and add instrumentation code to it statically — that is, without running the program. ATOM-based instrumentation tools work by inserting the instrumentation code into free space between segments, and inserting calls to that instrumentation at every relevant point in the executable. This is done to avoid having to modify offsets in the executable, which has already been fully linked and would be difficult to relocate again.

Kerninst. What ATOM does for user binaries, Kerninst does for a running Solaris kernel [46]. The tool takes object files containing instrumentation code and inserts them into the Solaris kernel, including any and all installed loadable modules, at runtime. This is done via a springboard mechanism: the assembly instruction before which instrumentation is to be inserted is placed at the beginning of a new piece of assembly code, which also includes the instrumentation function. A `ba,a` instruction, an unconditional branch that annuls the instruction in its delay slot, is inserted in its place; however, on the SPARC architecture, branches can only target code locations within 8 megabytes of the branch instruction. In most cases, there is not sufficient room to place the entire instrumentation function this close, but there is enough room to place a small "springboard," which contains a `call` instruction that jumps to the actual instrumentation function, and a `nop` for its delay slot.

2.2.3 Compiler-assisted debugging

Binary modification has two problems. First, inserting instrumentation code into a program is difficult to do without performing a full relocation afterward. This is because code is tightly packed and there is no room to insert code in-line, meaning that instrumentation code must be located outside the instruction stream, reducing performance and increasing implementation complexity. Second, instrumentation doesn't have access to all the rich type information and high-level structural information that the source code contains. Inserting instrumentation using the compiler solves both of these problems: it takes place before linking, and instrumentation tools have full access to the compiler's intermediate representations.

`gprof`. The `gprof` utility [19] post-processes data generated by code that has been instrumented using the GNU C compiler. The GNU C compiler can be instructed, via the `-pg` parameter, to instrument each time a function returns control to its caller, in such a way that they will increment a counter corresponding to the caller-callee pair. This produces a raw profile which is stored in an external file, `gmon.out`. The `gprof` tool uses this file to generate a call graph, identifying common calling sequences and cycles. This information is combined with information derived from program-counter sampling to produce a profile of the execution of the program.

2.2.4 Fixed tracepoints

Binary instrumentation and compiler-based instrumentation introduce significant implementation complexity. For some tasks, it is sufficient to simply modify the source code being instrumented by hand. This is particularly useful when instrumenting high-level events that can't be inferred by compilers.

Lockmeter. The Lockmeter utility is a profiler for spinlock access in the Linux kernel [6]. It consists of a patch that makes wrappers for the spinlock access macros in the Linux kernel. Each time the wrappers are invoked, the current program counter is recorded. Lockmeter maintains a hash table of program counters for locations where locks are taken and released, to which each location is added each time it is first seen. The addresses of the locks taken, and how long they were held, are stored in a separate array, and can be used to generate a profile of lock accesses. This profile can be used to find bottlenecks that limit SMP scalability. To speed up access to the lock array, which is a read-write data structure, independent versions are kept for each processor and data is only aggregated when necessary.

Linux Trace Toolkit. This system instruments a variety of events in the Linux kernel, and allows filtering and formatting of these events and their contexts before logging to a disk file [53]. It instruments a cross-section of kernel events: system call entry and exit, interrupts, events related to processes and the file system, VM and cache events, and networking and IPC events, among others. Like Lockmeter, the instrumentation consists of a patch to the Linux kernel source code. When instrumentation functions are invoked, a trace module is invoked. This trace module filters events based on event type, process, user, or group ID, and augments them with information such as CPU ID or the instruction pointer of the calling process (in the case of a system call). This information is stored in a buffer, which is periodically swapped with another buffer that is exposed to user logging processes via the `/proc` interface. The authors developed a graphical tool to visualize events, as well as showing all context switches between userspace and the kernel, as well as different userspace applications.

Chapter 3

Controlling Overhead

A central component of our proposed system is the mechanism that we use to ensure that overhead is predictable and controllable. As we will see in Section 3.3, overhead-control mechanisms typically fall into one of two categories. Mechanisms in the first category have a rate control that they autonomously reduce when particular targets are met. This rate control is not an overhead control; rather, it represents a best-effort method for reducing overhead. The second category attempts to minimize overhead using some non-quantitative approach.

In Section 3.1, we discuss our first approach, Monte Carlo Monitoring (MCM) which falls into the first category, and demonstrate Aristotle, a system based on it for detecting reference-counting bugs in the Linux kernel. In MCM, the goal is to observe some target number of events and then reduce the sampling rate. Then, in Section 3.2, we show our second approach, Software Monitoring with Controlled Overhead (SMCO), which adjusts monitoring to achieve a set overhead target. Our proposal is based on SMCO. Finally, in Section 3.3, we discuss other work in the area of overhead reduction.

3.1 Reducing Overhead as Confidence Increases

In this section, 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 using the instrumentation architecture discussed in Section 2.1. 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.

In previous work [20], Grosu and Smolka presented the MC^2 algorithm for *Monte Carlo Model Checking*. Given a (finite-state) reactive program P, a temporal property φ, and parameters ϵ and δ, 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, 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 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 verification*. 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 section is organized as follows. Section 3.1.1 describes our system design. Section 3.1.2 presents our Monte Carlo runtime monitoring algorithm. Section 3.1.3 details the Aristotle design and implementation. Section 3.1.4 gives an example application of Aristotle, and Section 3.1.5 contains our concluding remarks and directions for future work.

3.1.1 Aristotle Design Overview

Figure 3.1 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.

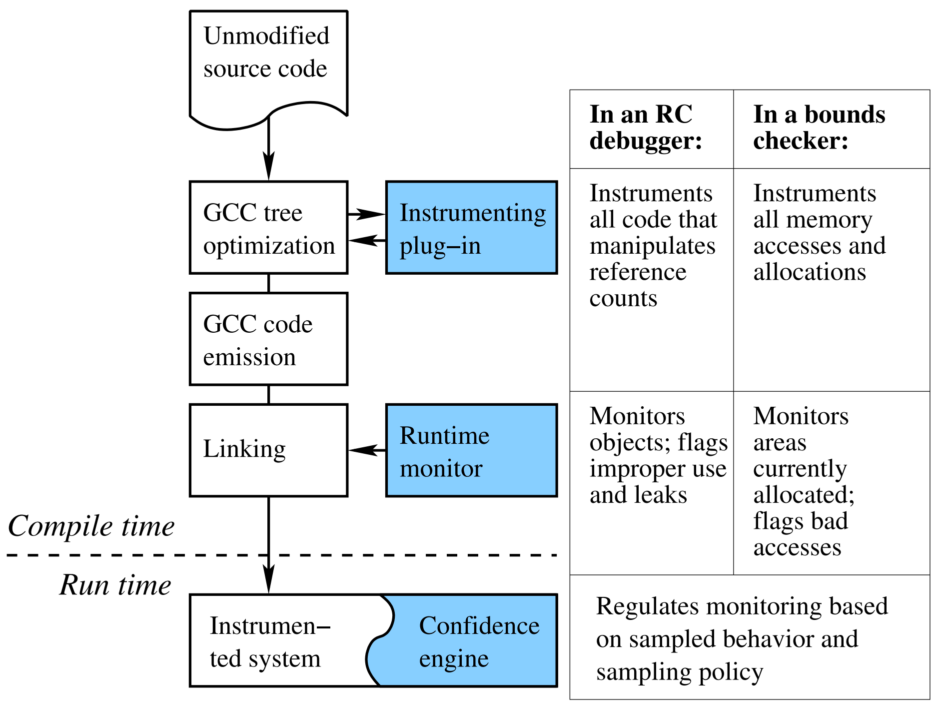


Figure 3.1: Architectural overview of the Aristotle system.

The runtime monitor interacts with the *confidence engine*, which implements a sampling policy based on our Monte Carlo runtime monitoring algorithm (described in Section 3.1.2). 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.

(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.

Figure 3.2: Reference-count correctness properties.

3.1.2 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 3.2.

Each of these properties is formalized using *Real-Time Linear Temporal Logic* [5], 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 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 [27, 49]. 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:

$$p_Z \geq \epsilon \;\Rightarrow\; \mathbf{P}[X \leq M] \geq 1 - \delta \tag{3.1}$$

Inequality 3.1 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* [34]. We define the *null hypothesis* H_0 as the assumption that $p_Z \geq \epsilon$. Rewriting inequality 3.1 with respect to H_0 we obtain:

$$\mathbf{P}[X \leq M \,|\, H_0] \geq 1 - \delta \tag{3.2}$$

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 3.3 which is obtained by taking the complement of $X \leq M$ in inequality 3.2:

$$\mathbf{P}[X > M \,|\, H_0] < \delta \tag{3.3}$$

With the above framework in place, we now present MCM, our Monte Carlo Monitoring algorithm. MCM, whose pseudo-code is given in Figure 3.3, utilizes DFA A to monitor properties stI, trI, and lkI, while keeping track of the number of samples taken.

```
input:   ε, δ, C, t, d;
global:  tn, cn;
tn = cn = ln(δ)/ln(1-ε);  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 */
```

Figure 3.3: 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 3.3, 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 3.3 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.

3.1.3 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 [18], 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 [7, 23] 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.

3.1.4 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.

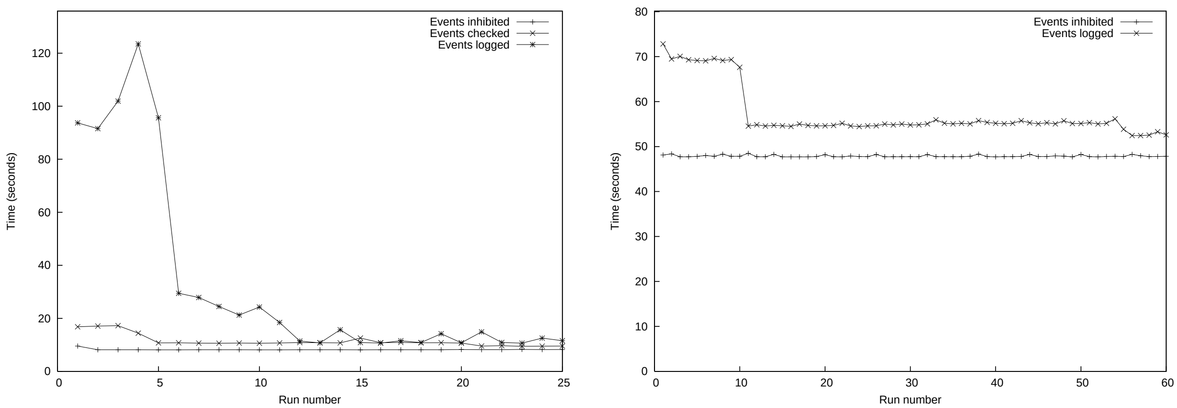
The fact that these caches are shared by different file systems, implemented by different authors and of varying degrees of maturity, introduces the potential for system resource leaks and faults arising from misuse of cached objects. For example, a misbehaving file system may prevent a storage device 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 3.1.2, 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 3.1.2, 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 3.4(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`

Figure 3.4: Overhead reduction as confidence increases.

Figure 3.4(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.

3.1.5 Conclusion

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.

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.

3.2 Bounding Overhead Using Supervisory Control

In this section, we introduce the new technique of *Software Monitoring with Controllable Overhead* (SMCO). SMCO is formally grounded in control theory, in particular, the *supervisory control of discrete event systems* [39, 1]. Overhead control, while maximizing confidence, is realized by disabling interrupts generated by the events being monitored—and hence avoiding the overhead associated with processing these interrupts—for as short a time as possible under the constraint of a user-supplied target overhead o_t. SMCO can be viewed as the problem of generating an optimal controller for a specific class of nonlinear systems that can be modeled as the composition of a set of timed automata. Our controller is designed in a *modular* way by composing a *global controller* with a set of *local controllers*, one for each monitored object in an application. Moreover, SMCO is a general monitoring technique that can be attached to *any* system interface or API.

We have applied SMCO to the problems of detecting *stale or underutilized memory* and checking for *bounds violations*. For memory staleness, we make novel use of the virtual memory hardware by utilizing the `mprotect` system call to protect each area suspected of being underutilized. If such an area is in fact accessed, the program generates a segmentation fault, informing the monitor that the area is not stale. If the time since the monitor protected an area is longer than a user-specified threshold, and there were no segmentation faults from that area, then it is stale. The SMCO controller controls the total overhead of memory under-utilization checking by enabling and disabling the monitoring of each memory area appropriately. For bounds checking, we use a plug-in–based approach as discussed in Section 2.1.

Experimental results of SMCO's performance on the `Lighttpd` Web server, the `vim` text editor, and our micro-benchmark suite are encouraging. SMCO maintains bounded overhead well for both applications. When local controllers behave linearly, SMCO controls overhead precisely; when local controllers behave non-linearly, it becomes more challenging for SMCO to control overhead, but results are still acceptable. For `Lighttpd`, we also demonstrate a discovery, that one-third of its heap footprint is completely unused. Our micro-benchmarks demonstrate how confidence grows monotonically with the target overhead up to CPU saturation, and that this is done consistently and predictably. Collectively, our benchmarking results show that using SMCO, it is indeed possible to achieve high-confidence monitoring with bounded overhead.

We organize the rest of this section in the following way. Section 3.2.1 explains SMCO's control-theoretic approach to bounding overhead while maximizing confidence. Section 3.2.2 presents our architectural framework for SMCO and describes how we apply it to bounds checking and staleness detection. Section 3.2.3 contains our performance evaluation and Section 3.2.4 offers concluding remarks and directions for future work.

3.2.1 Control-Theoretic Monitoring

The *controller design problem* attempts to regulate the input v to a process P, henceforth referred to as the *plant*, to make its output y adhere to a *reference input* x. The device that accepts x and y and produces v is called a *controller*; we write it Q. The composition of Q and P must make y approximate x with good dynamic response and small error (see Figure 3.5).

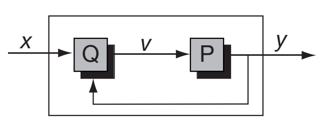


Figure 3.5: Plant (P) and Controller (Q) architecture.

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

The classic theory of digital control [13] assumes that the plant and the controller are linear systems. This assumption allows one to apply a rich set of design and optimization techniques, such as the Z-transform, fast Fourier transform, root-locus analysis, frequency response analysis, 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.

Because of the enabling and disabling of interrupts, the problem we are considering is nonlinear: 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 control* [39, 1].

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 [39, 1].

The monitoring overhead depends on the timing of events and the monitor's per-event processing time. The specification language L therefore consists of *timed words* $a_1, t_1, \ldots, a_n, t_n$ where each a_i is an (access) event and t_i is the time at which a_i has occurred. Consequently, the state machines used to model P and Q must also include a notion of time. Previous work has shown that supervisory control is decidable for *timed automata* [3, 52] and for *timed transition models* [40]. In our setting, we use a more expressive version of timed automata that allows clocks to be compared to variables, and for such automata decidability is not guaranteed. We therefore design our controller manually, but we are currently investigating techniques for the automated synthesis of an approximate controller. The controller we designed consists of the composition of a *global controller* and a set of *local controllers*, one for each plant (object in the application software) that we monitor. We define these two controllers later in this section.

Plant model.. We describe the plant P (see Figure 3.6) as an extended timed automaton whose alphabet consists of input and output events. We use timing constraints to label its locations and transitions. These constraints take the form $x \sim c$, where x is a *clock*, c is a natural constant or variable, and $\sim$ is one of $<$, $\leq$, $=$, $\geq$, and $>$. We write transition labels in the form `[guard] In / Out, Asgn`, where `guard` is a predicate over the automaton's variables; `In` is a sequence of input events of the form $v?e$ denoting the receipt of value e on channel v; `Out` is a sequence of output events of the form $y!a$ denoting the sending of value a on channel y; and `Asgn` is a sequence of assignments to the (local) variables. All fields in a label are optional. A transition is *enabled* when its guard is true and the event (if specified) has arrived. A transition is not *forced* to be taken unless letting time flow would violate the condition (invariant) labeling the current location.

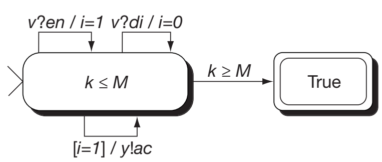


Figure 3.6: State machine for the plant P of one monitored object.

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

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

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

where the following conditions hold:

1. The average overhead $\overline{o} = (n\, p_a)/(t_n - t_1)$ is $\leq o_t$, where p_a is the average event-processing time.
2. If the strict inequality $\overline{o} < o_t$ holds, then the overhead undershoot is due to time intervals (with low activity) during which all access events are monitored.

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

The local controller.. Each monitored plant P has a local controller Q, the state machine for which is given in Figure 3.7. 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 Figure 3.8 for a graphical illustration of these intervals. Then $c_i = \tau_i + p_i + d_i$ is the total amount of time Q spends in the i-th cycle, and we refer to $o_i = p_i/c_i$ as the *overhead ratio* at i.

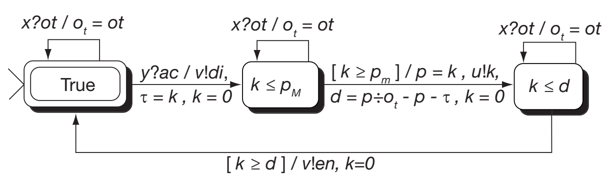


Figure 3.7: State machine for local controller Q.

To ensure that $o_i = o_t$ whenever the plant is throwing access events at a high rate, the local controller computes d_i as the least positive integer greater than or equal to $p_i/o_t - p_i - \tau_i$. If the plant throws events at a low rate, then all events are monitored and $d_i = 0$. Whenever processing of event $y?ac$ is finished, the local controller sends along u the processing time k to the global controller, which is discussed following the soundness and optimality proofs for the local controller. The processing time is assumed to lie within the interval $[p_m, p_M]$.

Soundness and optimality of the local controller.. We provide informal soundness and optimality proofs for our local controller.

Theorem 3.2.1 (Soundness)**.** *The language $L(Q\|P)$ of the parallel composition of the local controller Q with the plant P is included in the target specification language L.*

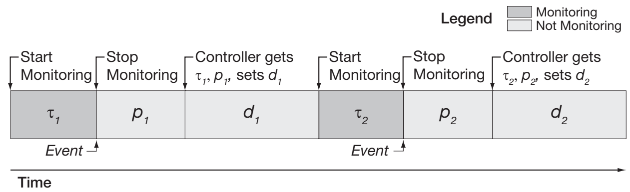


Figure 3.8: Timeline for local controller.

Proof. The soundness follows from the definitions of Q and P. If P throws events at a high rate, then Q maintains the mean overhead rate $\overline{o} = o_t$. If P throws events at a low enough rate, then Q monitors all events, and $\overline{o} < o_t$. If P alternates between intervals of high and low rates of event throwing, then $\overline{o} < o_t$; the difference between $\overline{o}$ and o_t is due to the low-rate intervals during which all events are monitored. □

The optimality condition we consider for the controller is with respect to the *space and time locality* of the plant. In particular, a useful property of our controller is that it tends to monitor a representative sample of "independent" events. This is because of its fine-grained control strategy; i.e., when the event rate is high, our controller (briefly) disables interrupts after *each* processed event. Since, by the space and time locality of the plant, consecutive events are likely to be similar or related, this strategy helps avoid monitoring similar events.

Theorem 3.2.2 (Locality-based optimality)**.** *The controller Q is optimal with respect to space and time* locality*.*

Proof. If an event is thrown at time t_i by a statement s or memory location m, then due to space locality, another event is likely to be thrown at a statement t or memory location n close to s and m, respectively. Therefore, the time t_j when the second event is thrown is also close to time t_i. Since Q disables interrupts immediately following occurrence of an event, optimality follows. □

The two applications of SMCO we consider are the detection of stale memory and bounds violations. For stale memory detection, once an event is thrown, we are certain that the corresponding object is not stale, so we can ignore interrupts for a definite interval of time, without compromising soundness and at the same time lowering the monitoring overhead. For array-bounds violations, we would like to analyze the program in a uniform way. Two bounds violations close to each other are likely to be caused by the same statement in the program. Hence, the first interrupt is enough to identify the bug, while also lowering the monitoring overhead.

The global controller.. The local controller Q achieves its target overhead o_t only if the plant P throws events at a sufficiently high rate. Otherwise the mean overhead $\overline{o}$ is less than o_t. In case we monitor a large number of plants P_i simultaneously, it is possible to take advantage of this under-utilization of o_t by increasing the overhead o_t of those controllers Q_i associated with plants P_i that throw interrupts at a high rate. In fact, we can scale the target overhead o_t of *all* local controllers Q_i with the same factor λ, as the controllers Q_j of plants P_j with low rate of interrupts will not take advantage of this scaling. Furthermore, we do this every T seconds, a period of time we call the *adjustment-interval*. The periodic adjustment of the local target overheads is the task of the global controller GQ. The architecture of our overall control framework for SMCO is shown in Figure 3.9.

The timed state machine for the global controller GQ is given in Figure 3.10. It inputs on x the user-specified target overhead ot, which it then assigns to local variable o_{gt} representing the global target overhead. It further outputs ot/n to the local controllers and assigns ot/n to local

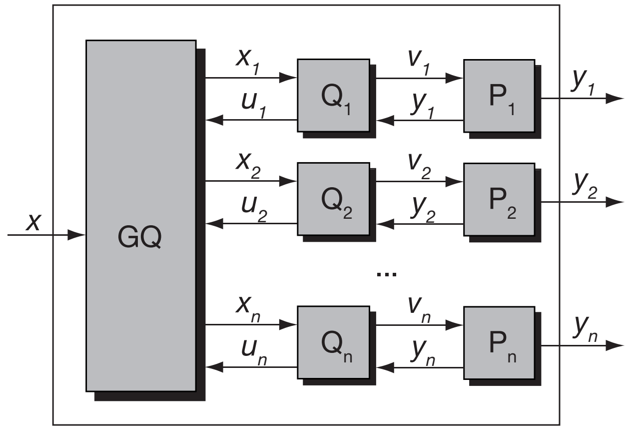


Figure 3.9: Overall control architecture.

variable o_t, representing the target overhead for the local controllers. The idea is that the global target overhead is evenly partitioned among the n local controllers. It also maintains an array of total processing time p, initially zero, such that $p[i]$ is the processing time used by local 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 local controller Q_i generates during the current adjustment interval.

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

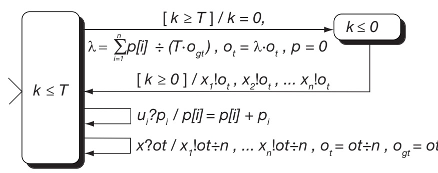


Figure 3.10: State machine for the global controller.

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

Theorem 3.2.3 (Global soundness)**.** *Let S be defined as $GQ \| Q_1 \| \ldots \| Q_n \| P_1 \| \ldots \| P_n$, the parallel composition of the global controller GQ, local controllers Q_i and plants P_i, $i \in [1..n]$. Then the language $L(S)$ is included in the target specification language L_G. Moreover, the discrepancy between $\overline{o}$ and o_{gt} is the minimal that can be achieved for the parallel composition of the plants and the adjustment interval of T seconds.*

Proof. We derive o_t so that $n \times o_t = o_{gt}$, where n is the number of plants. Each local controller Q_i achieves observed overhead $\overline{o_i} \leq o_t$, so $\sum_{i=1}^{n} \overline{o_i} \leq o_{gt}$. If the total is less than o_{gt}, then this is because some plants P_i are experiencing a low rate of interrupts, but in that case those plants have reduced their delays d_i to 0 so they are observing all possible events. Furthermore, because their rate is already as high as possible, under-utilized local controllers will be unaffected by λ-scaling, whereas others benefit. This fact can be used to prove minimal discrepancy. □

GQ also balances the load of local controllers in an optimal way with respect to the space and time locality of access events.

3.2.2 Design

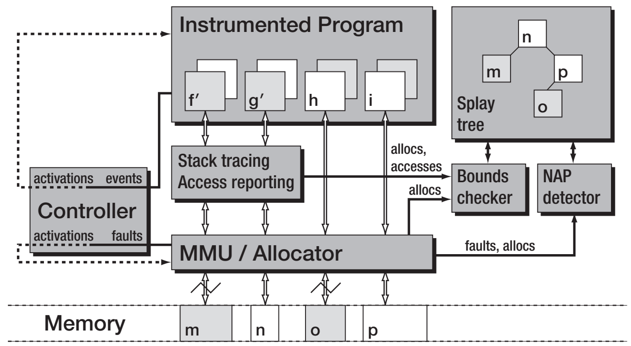


Figure 3.11: SMCO architecture for bounds checking and memory under-utilization detection.

In this section, we discuss the two applications that we have implemented for SMCO, namely memory under-utilization detection and bounds checking. An architecture overview of the system is shown in Figure 3.11. The *controller* in Figure 3.11 implements the global controller GQ and each local controller Q_i. The controller receives an event from the instrumented program each time the program executes a bounds check and from the Memory Management Unit (MMU) each time the program accesses a protected area. Based on the total time spent processing these events, the controller activates and deactivates monitoring of functions and memory areas.

The bounds checker and NAP (Non-Accessed Period) detector are responsible for detecting and reporting errors. Together, they maintain a splay tree of memory ranges, including stack areas and dynamic memory allocations. The bounds checker uses the splay tree to determine which accesses are out-of-bounds, and the NAP detector searches dynamic allocations in the splay tree to find regions that have not reported accesses in a prescribed amount of time. The *stack tracing/access reporting* module intercepts stack-area creations/destructions and pointer dereferences from instrumented functions and reports them to the bounds checker. Note that our architecture cleanly separates the overhead controller from the modules that perform fault detection and reporting.

Functions f and g and memory areas m and o in Figure 3.11 are rendered in gray to indicate that they are in the activated state. Function calls to f or g will therefore result in the execution of their instrumented versions f' and g', respectively, so that bounds checking, with controllable overhead, can be performed. Similarly, the MMU will intercept accesses to m and o so that events can be generated for processing by the NAP detector.

We now describe the controller, the NAP detector, and the bounds checker in more detail.

Controller Design. The controller's role is to limit the number of events generated by the instrumentation in order to meet a target overhead goal, where *overhead* refers to the percentage of time an instrumented program spends processing the events. To this end, after every event, the controller temporarily disables events from the entity (activated function or memory area, in our case) that generated the event. Consider bounds checking. A memory access within a function call generates a bounds-checking event. The controller measures the total amount of time the bounds check takes, and then computes a delay using the function's *local controller*, as described in Section 3.2.1. The function runs with bounds checking turned off until the delay passes and the controller reactivates it. A similar mechanism temporarily deactivates a protected memory area when it generates an access event.

NAP **Detection.** We have implemented an SMCO-based under-utilization detector which identifies areas that are not accessed for a user-definable period of time. We refer to such a time period as a *Non-Accessed Period*, or NAP. Figure 3.12 depicts the error model for the under-utilization detector. Note that we are not detecting areas that are never touched (i.e., leaks), but rather areas that are not touched for a sufficiently long period of time to raise concerns about memory-usage efficiency.

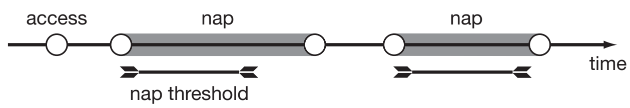


Figure 3.12: Our memory under-utilization detector reports non-accessed periods. NAPS *can vary in length, and multiple* NAPS *can be reported for the same area.*

The implementation of a memory under-utilization detector would normally involve instrumenting memory-access instructions. This technique, however, introduces a number of issues, both practical and theoretical. First, there is the practical issue of finding accesses. This can be done using a compiler or by using tools like Valgrind; compiler tools only works for programs with available source, and both introduce significant overheads. Our under-utilization detector controls its overhead using sampling. However, if one attempts to find NAPS by sampling a subset of memory-access instructions, this introduces a theoretical problem: in order to be able to say with certainty whether or not a particular area was accessed in a particular time period, *all* memory-access instructions must be monitored during that period, which could again result in considerable overhead.

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

To perform our memory-access interposition, we implemented a shared library that replaces the standard memory-allocation functions, notably `malloc` and `free`, with functions that handle memory in multiples of the block size that `mprotect` can protect. Due to the implementation of memory protection in hardware, this block size is nearly always larger than the smallest addressable unit: on x86-based platforms, it is usually 4096 bytes. After allocating an area, our custom allocator adds an entry to a splay tree that contains information about its size, the last time an access was observed, and data to support our controller, which controls each area individually.

When the controller instructs `memcov` to monitor an area, `memcov` uses `mprotect` to disallow reads and writes that reference that area. Then, when the program attempts to access that area, the memory protection hardware intercepts the access and the kernel passes a `SIGSEGV` to the process. The signal is then handled by `memcov`, which performs a lookup in the splay tree and registers a hit with the appropriate controller. Memcov periodically checks for areas that have been monitored and unaccessed for longer than the user-defined threshold, and reports them as NAPs if that NAP has not already been reported.

Bounds Checking. Our second application is a more traditional problem: bounds checking. Bounds checking may be broadly defined as ensuring that pointers are dereferenced only when they are *valid,* which typically means that they point to memory addresses located in properly-typed regions of the stack, heap, or static (including text, data, and BSS) segments of the program's address space. Our definition of a valid pointer is one that points to a region that

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

We consider any dereferenced pointer to be valid if its target matches the above criteria, regardless of the pointer's type or the region it originally pointed to. This means that we do not need to keep track of each pointer update, which would impose additional overheads. Instead, we need only keep track of areas as they are allocated and deallocated. To accomplish this, we use the splay tree that we use for NAP detection. At the entry to each function, the function's stack variables and static variables are registered in the splay tree. At each function's exit points, the function's stack variables are deregistered.

To add instrumentation to a program, we use a branch of the GNU C compiler modified to use plug-ins [8]. Plug-ins are written as normal GCC-optimization passes that modify GCC's GIMPLE intermediate representation, but can be compiled separately from GCC and loaded dynamically. This dramatically reduces turnaround time for modifications and facilitates debugging. Our bounds-checker plug-in, called `meminst`, performs three tasks, which we discuss in detail below: emitting registrations and deregistrations, duplicating the source code for each function, and emitting instrumentation into one of the copies. Figure 3.13 shows an example duplicated function along with its added instrumentation and the control block that switches between the two copies.

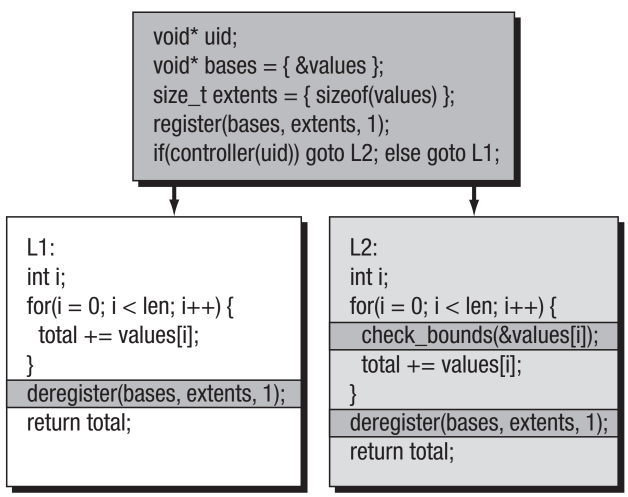


Figure 3.13: `meminst` *adds initial registrations and a call to the controller in a function's first block; the rest is duplicated, and one copy (left) of the function only has deregistrations, whereas the instrumented copy (right) also includes bounds checking.*

Emitting registrations/deregistrations.. `meminst` first locates each addressable variable in the internal representation of the function being transformed, and adds it to a list. This takes $O\left(n + r^2\right)$ time, where n is the size of the function's IR and r is the number of such variables (the r^2 is due to the fact that we enforce uniqueness in the list). It then builds an array containing the address of each variable, and another array containing the size of each variable. The plug-in adds a call to an area-registration function at the beginning of the function, and a call to an area-deregistration function at each return point from the function.

Duplicating the source code.. To create instrumented and uninstrumented versions of the program, `meminst` duplicates the basic blocks in the control-flow graph for each function. This takes $O(n)$ time, where n is the size of the function's internal representation. In order for the controller to determine which set of basic blocks is executed before the beginning of each function, `meminst` inserts a call to the controller. The controller maintains a data structure corresponding to each function, which contains the computed τ for that function and the most recent value of p; `meminst` adds a static variable to the function that is passed to the controller and which the controller sets to point to this structure.

Emitting instrumentation.. At each pointer dereference or array-member access in the instrumented copy of the code, `meminst` adds a call to the bounds checker. This step takes $O(n)$ time.

3.2.3 Evaluation

In this section, we describe a series of benchmarks we ran to validate our implementation and determine its runtime characteristics. The results show that SMCO fulfills its goals: it closely adheres to the desired overhead for a range of target overheads, and it observes events at higher rates, and catches bugs with greater effectiveness, as more overhead is allowed. We begin with a real-world demonstration using the Lighttpd Web server [26] and the Vim 7.1 text editor [50]. Then we further investigate the effectiveness of SMCO by demonstrating its usage with a microbenchmark that causes bounds violations.

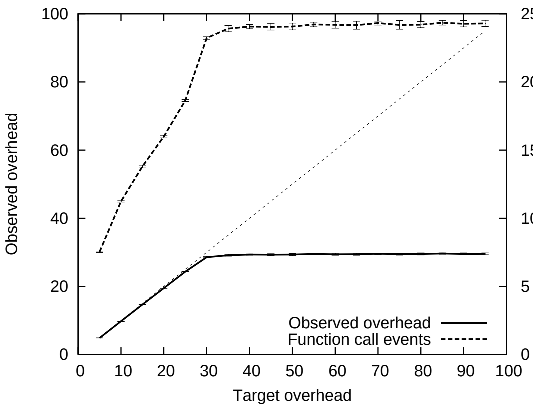
We ran our benchmarks on a group of identically configured machines, each with two 2.8GHz EM64T Intel Xeon processors with 2 megabytes of L2 cache each. The computers each had 1 gigabyte of memory and were installed with the Fedora Core 7 distribution of GNU/Linux. The installed kernel was a vendor version of Linux 2.6.23. We built all packages tested from source: we built the instrumented programs with a custom 4.3-series GCC compiler modified to load plugins [8], and we built other utility programs using a vendor version of GCC 4.1.2. Our Lighttpd benchmarks use Lighttpd version 1.4.18. Graphs that have confidence intervals show the 95% confidence interval over 10 runs, assuming a sample mean distributed according to the Student's-t distribution.

Overhead Control Benchmark Results

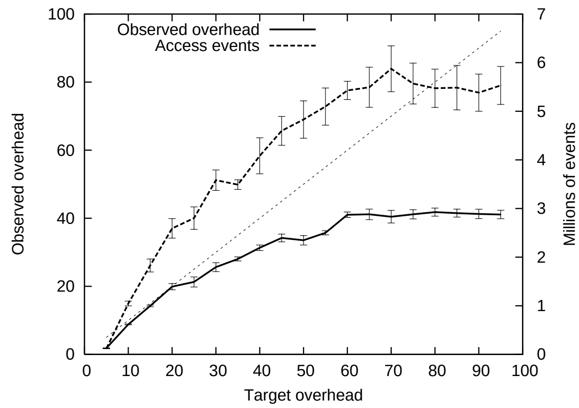
SMCO's main goal is to monitor as much as possible while regulating overhead so that it adheres closely to the specified target overhead. This adherence should be largely independent of the load conditions of the system. As our theoretical result in Section 3.2.1 shows this to be achievable, any deviation of the measured performance results from user specification must arise from implementation limitations.

The first and most obvious limitation is saturation: at high enough overhead, events stay enabled all the time. Increasing the desired overhead past this point will not generate more events to process. All programs eventually reach peak overhead, since a program must spend at least some percent of its time outside of event processing in order to generate events. A less obvious limitation is the latency of global controller updates. Changing conditions can cause overhead increases that require a quick response from the global controller; when the target overhead is low and the load is high, the actual overhead may temporarily exceed the target overhead.

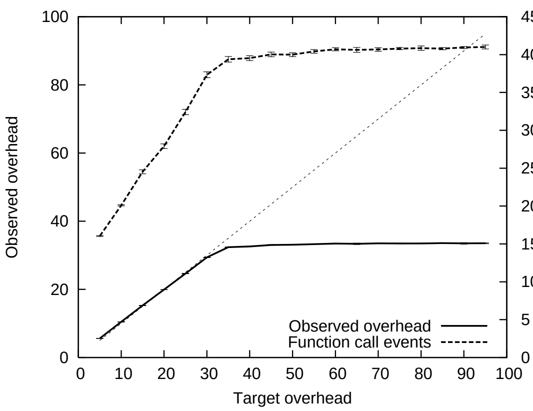
Figure 3.14 shows observed overhead vs. target overhead for the Lighttpd server and the Vim



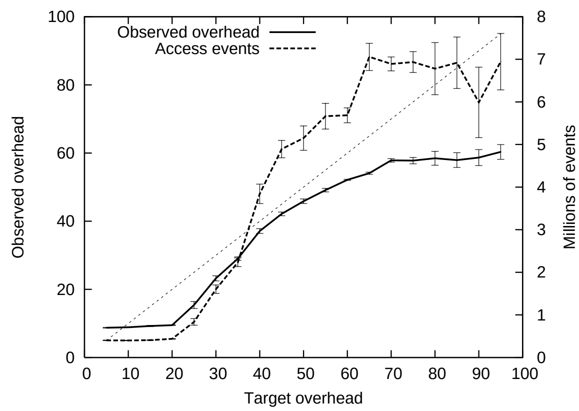
(a) Bounds checker on Lighttpd



(b) Memory under-utilization detector on Lighttpd



(c) Bounds checker on Vim



(d) Memory under-utilization detector on Vim

Figure 3.14: Observed load versus desired load for the Lighttpd server with 75 clients issuing one request per second and the Vim text editor with an automated workload. Observed overhead is shown in % (y axis), and the number of observed functions calls (for the bounds checker) and memory accesses (for the under-utilization detector) are shown in millions (y2 axis).

text editor, each tested separately with bounds checking and memory under-utilization detection. For Lighttpd, we use the `curl-loader` tool to hit the server with one request per second from each of 75 simulated clients. We ran Vim with a scripted workload that loads a large (1.2MB) text file and alternates between sorting and reversing its contents. We ran Lighttpd and Vim with target overheads from 5% to 100% in increments of 5%. The solid line shows the observed percent overhead (left axis), which should ideally adhere to the thin dotted $y = x$ line. The dotted line shows the number of events processed—function-call events for the bounds checking benchmark and memory-access events for the under-utilization detector—in millions of events (right axis).

For the bounds checker, the observed overhead closely tracks the target overhead up to a target overhead of 30%. At higher target overheads, the system saturates: every function call runs with bounds checking on, leaving no opportunity to produce higher overhead.

In the Lighttpd memory under-utilization results, we observe that the system meets its overhead target in the region from 5% to 20%. In the region from 20% to 60%, the local controllers start to show instability. After deactivation, each memory area waits for reactivation on a priority queue. In the unstable region, memory regions spend most of their time on this priority queue. When the queue size gets large, dequeueing a memory region takes longer, adding to its wait time and preventing it from meeting its overhead goal. This effect is non-linear—changing the desired overhead changes the average queue size—so our global controller does not accurately compensate for it, though the system is able to keep actual overhead below the bound in all cases.

Above 60% target overhead, the system achieves its maximum possible overhead: memory regions skip the priority queue and reactivate as fast as possible. The Vim under-utilization results also show instability, with the global controller unable to use all the overhead it is allowed, up to 70% desired overhead.

During our benchmarking, we did not observe any bounds violations resulting from bugs in Lighttpd. However, we observed a number of NAPs (non-accessed periods, see Section 3.2.2). This can be seen in Figure 3.15, which shows the proportion of areas that have been monitored for various periods of time over a single run of Lighttpd. Unused regions in the run were classified into eight buckets based on how long the region remained active with no access. Each set of stacked vertical bars shows the number of bytes in regions from each bucket at a time during Lighttpd's run, with the lightest shaded bar showing the allocations that have gone longest without an access.

This graph shows that a comparatively large amount of memory goes unused for almost all of the program's run. Lighttpd is intended as an embedded Web server with a low memory footprint. Its total heap footprint is 540 kilobytes—so the unused 180 kilobytes are of particular interest and comprise a significant reduction in Lighttpd's heap memory footprint (one-third less). We have verified that at least some of these areas come from a pre-loaded MIME type database that could be loaded incrementally on demand.

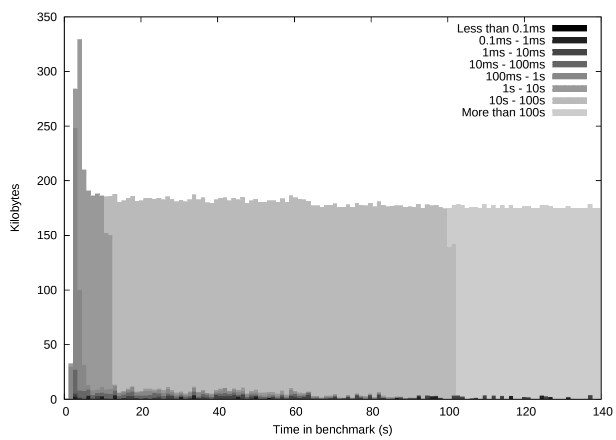


Figure 3.15: Age of areas in one run of Lighttpd. Each stack of bars represents one point in time under the medium load benchmark. The individual bars correspond to ages of areas; their size indicates the number of kilobytes of areas that have that age.

Micro-Benchmark Results

Having demonstrated the effectiveness of SMCO with real-world applications, we turn to the micro-benchmarks to demonstrate the high-confidence nature of SMCO. Specifically, we demonstrate that an SMCO-based monitor will detect more faults in a buggy system when it is allowed to use more overhead.

We first designed a micro-benchmark called MICRO-BOUNDS that runs for ten seconds, accessing a single memory area as fast as it can. Ten times per second, it issues an out-of-bounds access. This micro-benchmark allows us to examine the performance of SMCO in more detail.

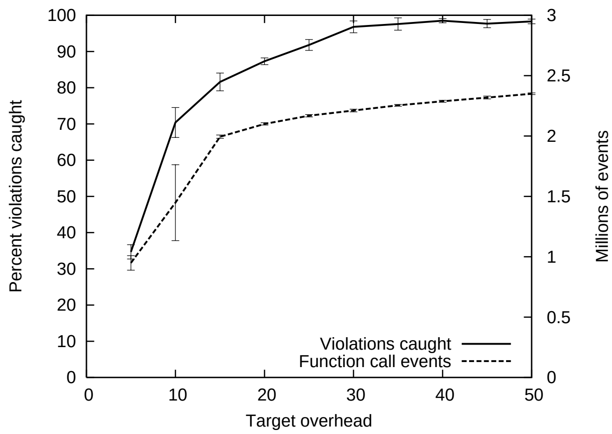


Figure 3.16: Number of function executions bounds-checked and number of bounds violations caught, versus target overhead for the MICRO-BOUNDS *micro-benchmark.*

Figure 3.16 shows our effectiveness at detecting bounds violations in this micro-benchmark for different target overhead settings. The solid line shows the percent of bounds violations caught, and the dotted line shows how many events SMCO observed overall. (Under-utilization detection is still active, but we do not show the results because MICRO-BOUNDS does not use heap-allocated memory.) Initially, we observe a linearly increasing number of accesses, which saturates near 100% of accesses observed, confirming that we are not only achieving our overhead targets, but we are in fact getting something for that overhead: we are monitoring as much as possible given the overhead constraints.

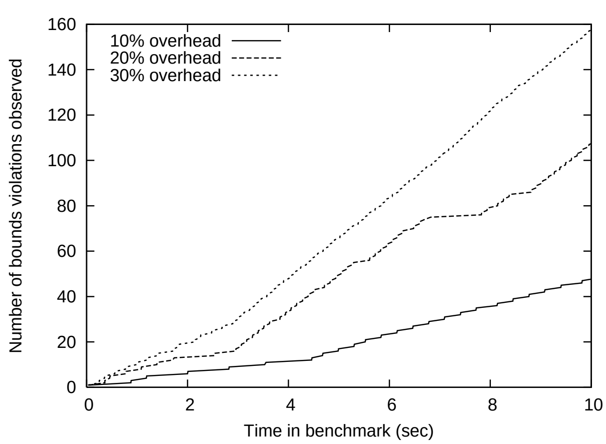


Figure 3.17: Cumulative number of bounds violations caught over time in runs of the MICRO-BOUNDS *micro-benchmark with 10%, 20%, and 30% overhead*

Figure 3.17 demonstrates the bounds-checking characteristics of our micro-benchmark over time with different target overheads. Each line shows a single 10 second run of the micro-benchmark, with the y-values being the total number of bounds violations observed up until each point in time. The graph demonstrates that the SMCO-based monitor continues to observe bounds violations at a roughly uniform rate. This demonstrates the difference between SMCO and most existing adaptive sampling tools [24], which reduce overhead over time, making violations occurring later less likely to be caught than early violations.

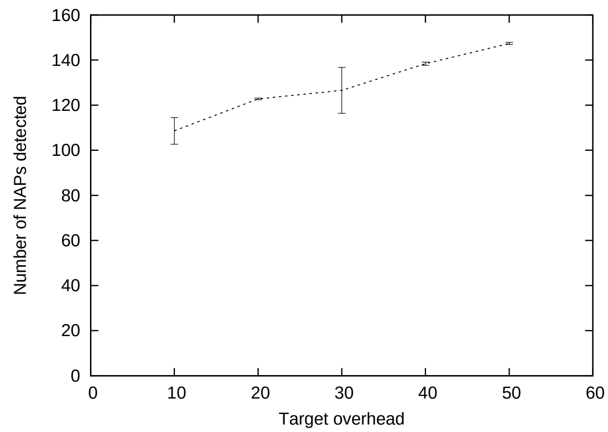


Figure 3.18: Observed NAPs *increase with target overhead for the* MICRO-NAP *micro-benchmark.*

Finally, we designed a micro-benchmark called MICRO-NAP that runs for one minute. It maintains 100 separately-allocated heap areas—allowing the benchmark's auxiliary data structures to reside comfortably in CPU caches—and uses a pseudo-random number generator to generate access intervals for each of these areas. Initially, and whenever it performs an access, MICRO-NAP generates a random interval between one and eight seconds (a NAP is three seconds, and we left time for monitoring to resume); after this interval, it performs an access. Figure 3.18 shows that given more overhead, the NAP detector finds more NAPs. We observe that NAP detection works well even with low overheads, because underutilized memory areas require little overhead to monitor so their local controllers can afford to monitor them regardless of global targets.

The MICRO-BOUNDS and MICRO-NAP micro-benchmarks demonstrate that both bounds checking and NAP detection can use additional overhead effectively.

Summary of Experimental Results

In this evaluation section, we have demonstrated two things:

1. SMCO is zealous in enforcing the overhead goal specified by the user while making a best effort to observe increasing amounts of events, and
2. tools using SMCO are effective at detecting memory under-utilization and bounds violations.

When the local controllers behave linearly, SMCO maximizes the number of events it observes by using as much overhead as it is allowed. With non-linear local controllers, the overhead control problem is more difficult, but SMCO still enforces an upper bound on overhead. We have also shown that SMCO-based monitoring tools observe more events as they are given more overhead, and can use this information to catch real as well as injected errors. In Lighttpd, we found that 33% of the heap footprint is spent on memory that is completely unused, even under load.

3.2.4 Conclusion

We have presented Software Monitoring with Controllable Overhead (SMCO), an approach to overhead control for the runtime monitoring of instrumented software. SMCO is high-confidence because, as we have shown in the paper, it monitors as many events as possible without exceeding the target overhead level. This is distinct from other approaches to software monitoring which 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 an underlying control strategy based on an optimal controller for a nonlinear control problem represented in terms of the composition of timed automata.

Using SMCO as a foundation, we have developed two sophisticated monitoring tools: a memory staleness detector and a bounds checker. The staleness detector detects memory areas that are unused for longer than a user-definable interval. This is achieved by taking advantage of memory-protection hardware, a technique normally used in the kernel to evict pages from physical memory but rarely seen in user-space. The bounds checker instruments memory accesses and checks them against a splay tree of valid areas. Both the per-area checks in the staleness detector and the per-function checks in the bounds checker are activated and deactivated by the same generic controller, which achieves a desired target overhead with both of these systems running.

Our benchmarking results demonstrate that it is possible to perform correctness monitoring of large software systems with fixed overhead guarantees. As such, the promise of SMCO is attractive to both developers and system administrators; developers desire maximal monitoring coverage, and system administrators need a way to effectively manage the overhead runtime monitoring imposes on system performance. Moreover, our system is fully responsive to changes in system load, both increases and decreases, which means that administrators need not worry about load spikes causing unusual effects in instrumented software. We also demonstrated the effectiveness of our system at detecting real-world bugs: for example, we found out that one-third of the Lighttpd Web server's heap footprint is unused.

3.3 Other Approaches to Controlling Overhead

In this section, we discuss approaches to overhead control that others have implemented, and which provide context for our own approach.

Java-based code replication. Arnold and Ryder developed a system that performs profiling by replicating code using the Java just-in-time (JIT) compiler [4]. This was implemented as a compilation phase for the Jikes research virtual machine [48]. In this approach, instrumentation activates when a counter expires. This is similar to couter overflow sampling, which we discussed in Section 2.2.1. This gives instrumentation developers a knob to control instrumentation, adding another input to the system. However, it does not provide any feedback mechanism except benchmarks, and consequently is not suited to inline control.

Context-based overhead reduction. Artemis reduces overhead from runtime checks by enabling them for only certain function executions [12]. To observe as many behaviors as possible, Artemis always monitors a function when the function runs in a *context* that it has not seen before, where a function's execution context consists of the values of global variables and arguments. Artemis's context-based filtering can, when the developer specifies the contexts correctly, be very effective for focusing instrumentation on areas where it is needed. Artemis, however, does not address the case where the target overhead is prescribed by the user. Also, Artemis does not address timing issues because timing is not part of an Artemis context.

Leak detection using adaptive profiling. SWAT is a leak-detection tool that uses a binary instrumentation framework to periodically rewrite pieces of code, enabling monitoring for those pieces of code [24]. The monitoring observes the memory accesses those code fragments make, and use that information to infer which memory areas are unused. SWAT reduces the rate at which it activates individual regions as they execute, meaning that commonly-used codepaths are monitored less intensively. This means that commonly-used pieces of code use less overhead. This approach resembles `MCM`, which we discussed in Section 3.1 in that it keeps monitoring high for rarely-used code, but there is no formal definition of confidence used.

Chapter 4

Research Plan

In this thesis proposal, we have two main goals. First, we intend to create a system that allows programmers to insert arbitrary debugging code into an application after deployment, while properly respecting users' performance requirements. Second, we intend to leverage the compiler to provide more detailed semantic information at each possible point where the programmer could add instrumentation, and to provide instrumentation points in a more intelligent manner. The general architecture of the system we intend to develop to achieve these goals is shown in Figure 4.1.

While the networked aspect of our system is important for future implementation, we intend to make the system as amenable as possible to network-extension without addressing the problems of scale and management that have been addressed by existing cooperative debugging research [29].

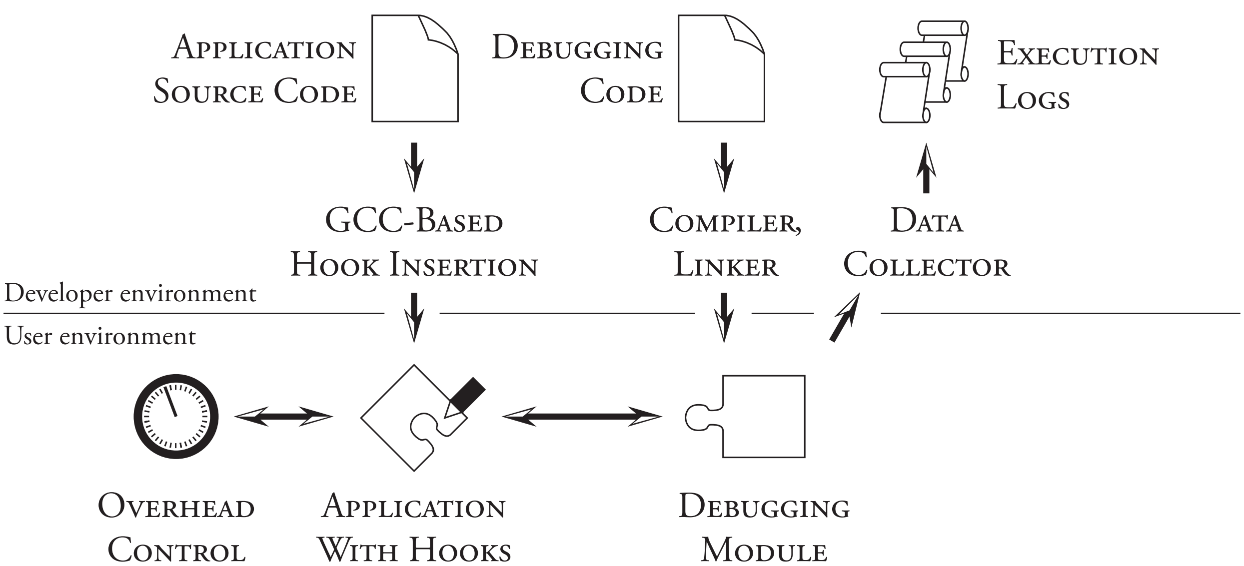


Figure 4.1: Architecture of the proposed system.

GCC will insert instrumentation points into the application's source code, both automatically and in response to user annotations. By default, GCC will insert instrumentation at each function entry point. If annotations do not indicate anything else, instrumentation will be provided with the arguments that the function receives, and with the values of static variables in the function.

Annotations will be able to specify not only code locations but also data structures which, when they are accessed, can trigger instrumentation.

We will also explore what other data may be useful (such as the function's locals), as well as how to provide access to these and other internal variables if instrumentation requests them. We have already been able to expose a program's internal state to instrumentation in the case of GCC, by modifying its link process to expose its global variables as symbols to plug-in code.

When executing, the application will load instrumentation out of a plug-in file, specified as an environment variable. This is intended to simplify future enhancement, where the code is downloaded from a server. The instrumentation will specify where it needs to execute, and the application's instrumentation hooks will collaborate with an overhead control mechanism to ensure that instrumentation only uses a fixed proportion of the program's execution time. The instrumentation overhead has several components, which we will quantify individually.

- Disabled instrumentation points incur a constant overhead per function execution. We will determine this overhead by storing the time at the beginning and end of each tracepoint in a pair of static variables. The difference will be calculated during the next execution of the tracepoint, to allow the overhead from that calculation to be measured as well.
- The overhead from execution of instrumentation functions can be measured accurately by calculating the time before and after the call.
- The time spent calculating the next activation time, and maintaining auxiliary data structures, has been a source of overhead in the past. We will ensure that all computations related to the overhead-control mechanism itself are themselves profiled.

Initially, the instrumentation will write its output to a local log file. We will investigate ways to optimize this so as to minimize file-access overhead, as well as providing upload functionality to the instrumenation server.

We will execute this project in several phases:

1. Develop a GCC plug-in that loads always-on instrumentation into a program and accurately reports the overhead impact of that instrumentation on the program. Benchmark this.
2. Modify the GCC plug-in to bound the instrumentation's overhead as discussed in Section 3.2. Benchmark this.
3. Add functionality to GCC supporting user-specificable tracepoints, both at specific points in the source code and also attached to specific variables.
4. Separate the instrumentation into loadable modules, and provide a uniform logging API.

The deliverable will be a tool that is capable of accomplishing the above on C code, with tests on the Web servers, text editors, FTP servers, standard benchmarks like SPEC, among other C applications. We will leave the following as future work:

- Designing the client/server system that obtains new instrumentation code and reports results.
- Extending the system to support C++.
- Adding code signing and encryption to support secure debugging.

Bibliography

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

[2] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer USENIX Technical Conference*, pages 93–112, Atlanta, GA, June 1986. USENIX Association.

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

[4] M. Arnold and B. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, Salt Lake City, UT, June 2001.

[5] A. Bernstein and P. K. Harter. Proving real-time properties of programs with temporal logic. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP 1981)*, pages 1–11, Pacific Grove, California, 1981. ACM Press.

[6] R. Bryant and J. Hawkes. Lockmeter: Highly-informative instrumentation for spin locks in the Linux kernel. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 271–282, Atlanta, GA, October 2000. USENIX Association.

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

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

[9] B. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 15–28, 2004.

[10] Silicon Graphics Corporation. *SpeedShop User's Guide*. SGI Technical Publications, Mountain View, CA, USA, 2003.

[11] David MacKenzie and Ben Elliston and Akim Demaille. Autoconf: Creating automatic configuration scripts. `www.gnu.org/software/autoconf/manual/autoconf.pdf`, 2006.

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

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

[14] Free Software Foundation. Shared library support for gnu. `www.gnu.org/software/libtool/manual.html`, 2005.

[15] The Free Software Foundation, Inc. GDB: The GNU Project Debugger. `www.gnu.org/software/gdb/gdb.html`, January 2006.

[16] G. C. Necula and S. McPeak and S. P. Rahul and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, England, 2002. Springer-Verlag.

[17] M. Galpin. Developing applications using the Eclipse C/C++ development toolkit. `www.ibm.com/developerworks/opensource/library/os-eclipse/stlcdt/index.html`, 2007.

[18] The GCC team. *GCC online documentation*, December 2005. `http://gcc.gnu.org/onlinedocs/`.

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

[20] R. Grosu and S. Smolka. Monte carlo model checking (extended version). In *Lecture Notes in Computer Science*, volume 3440, pages 271–286. Springer Verlag, 2004.

[21] Free Standards Group. DWARF debugging information format, version 3. `dwarfstd.org/Dwarf3.pdf`, December 2005.

[22] M. Haardt and M. Coleman. *ptrace(2)*. Linux Programmer's Manual, Section 2, November 1999.

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

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

[25] Apple Inc. *Xcode User Guide*. Apple Inc., Cupertino, California, 2008.

[26] Jan Kneschke. Lighttpd. `http://www.lighttpd.net/`, 2003.

[27] O. Kupferman and M. Y. Yardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

[28] F. E. Levine. A programmer's view of performance monitoring in the PowerPC microprocessor. *IBM Journal of Research and Development*, 41(3), 1997.

[29] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '03)*, San Diego, CA, June 2003.

[30] M. Boshernitsan and S. L. Graham. Interactive transformation of Java programs in Eclipse. In *Proceedings of the 28th Internation Conference on Software Engineering*, pages 791–794, New York, NY, 2006. ACM Press.

[31] R. McDougall and J. Mauro. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture, Second Edition*. Prentice Hall, Upper Saddle River, New Jersey, 2006.

[32] R. McDougall, J. Mauro, and B. Gregg. *Solaris Performance and Tools*. Prentice Hall, Upper Saddle River, New Jersey, 2007.

[33] J. Merrill. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In *GCC Developers Summit*, 2003.

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

[35] D. Novillo. TreeSSA: A New Optimization Infrastructure for GCC. In *Proceedings of the 1st GCC Developers' Summit*, Ottawa, Canada, May 2003.

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

[37] The GDB Project. STABS. `sources.redhat.com/gdb/onlinedocs/stabs.html`, 2004.

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

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

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

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

[42] A. Singh. *Mac OS X Internals: A Systems Approach*. Addison-Wesley, Upper Saddle River, New Jersey, 2007.

[43] M. Snyder and J. Blandy. The Heisenberg debugging technology. In *Proceedings of the 1999 Embedded Systems Conference*, San Jose, California, September 2007. `sourceware.org/gdb/talks/esc-west-1999/paper.pdf`.

[44] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. *SIGPLAN Not.*, 39(4):528–539, 2004.

[45] Sun Microsystems, Inc. *dbx man page*. Sun Studio 11 Man Pages, Section 1.

[46] A. Tamches and B. P. Miller. Using dynamic kernel instrumentation for kernel and application tuning. *The International Journal of High Performance Computing Applications*, 13(3):263–276, Fall 1999.

[47] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[48] The Jikes RVM Project. Jikes RVM. `jikesrvm.org`, 2007.

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

[50] *Vim 7.1*, 2007. `www.vim.org`.

[51] J. Wetzel, E. Silha, C. May, B. Frey, J. Fukukawa, and G. Frazier. *PowerPC Operating Environment Architecture, Book III*. IBM Corporation, Austin, Texas, 2003.

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

[53] K. Yaghmour and M. R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *Proc. of the Annual USENIX Technical Conference*, pages 13–26, San Diego, CA, June 2000. USENIX Association.

[54] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance analysis using the MIPS R10000 performance counters. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, November 1996.