A Lightweight Architecture for Program Execution Monitoring*

Clinton Jeffery, Wenyi Zhou, Kevin Templer and Michael Brazell
(jeffery|wzhou|ktempler|mbrazell)@cs.utsa.edu
Division of Computer Science, University of Texas at San Antonio

Abstract

The Alamo monitor architecture reduces the difficulty of developing dynamic analysis tools, such as special-purpose profilers, bug-detectors, and program visualizers.

1 Introduction

Dynamic analysis tools are used in several phases of software development, including coding, testing, and maintenance [1]. Although conventional debuggers and profilers are well-suited for finding certain kinds of bugs and performance bottlenecks, they may be ineffective when problems arise for which they were not intended.

Improvements in execution monitors have been slow to appear, primarily due to the high cost of developing such tools. This motivates the focus of our research: reducing the cost of writing monitors. A monitor framework for the Icon programming language presented one approach that reduces development costs for a broad class of execution monitors [2]. That framework provided monitor writers with solutions for several problems inherent in the execution monitoring realm, such as access to and control of another program's execution, and efficient techniques for dealing with the large amount of information to be processed. Because the framework was developed for an interpreted virtual-machine language implementation, the applicability of these results was limited to similar interpretive language implementations.

The Alamo monitor architecture extends and generalizes the work done for monitoring in the Icon interpreter by adapting the execution model and developing implementation techniques suitable for monitoring compiled programs. *Alamo* stands for A Lightweight Architecture for MOnitoring. The Alamo architecture consists of (1) an automatic instrumentation mechanism, (2) an execution model, (3) abstractions for event selection, multiplexing and composition, and (4) an access library that allows monitors to directly manipulate target program state. These four components are applicable to many compiled and interpreted languages. Figure 1 gives an overview of the Alamo architecture.

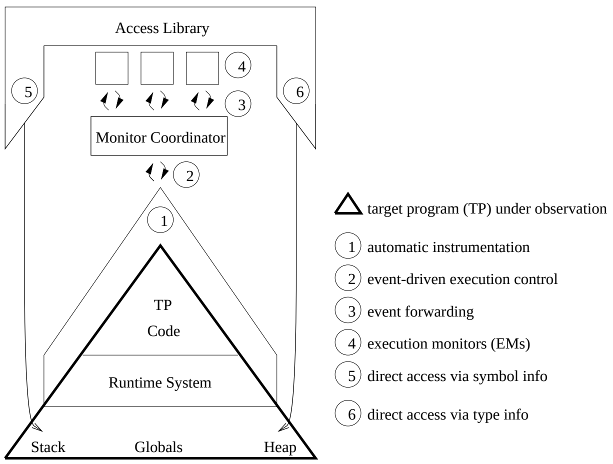

Figure 1: The Alamo architecture.

The techniques used to implement Alamo for an interpretive language are fairly easy compared with those required for a compiled language; the earlier Icon framework, with some refinements to the event selection mechanism, is an instantiation of the Alamo architecture for a language interpreter. The main emphasis in Alamo has been development of techniques for monitoring compiled programs. In order to prove the applicability of the Alamo architecture to compiled languages, an Alamo framework has been developed that reduces the cost of writing monitors for ANSI C programs.

This paper presents the Alamo architecture and techniques developed for the monitoring of compiled C code. The Icon framework has proven to be a useful testbed for the C framework implementation, as well as a prototyping environment where monitors can be developed and tested, prior to their subsequent C imple-

*This work was supported in part by the National Science Foundation under Grant CCR-9409082.

A revised version of this paper will appear in the 1998 ACM SIGPLAN Workshop on Program Analysis for Software Tools and Engineering Montreal, Canada, 14 June 1998.

mentation. Together, the Alamo Icon framework and the Alamo C framework also provide a design and a collection of implementation techniques that can be applied to monitoring other languages.

The implementation of the Alamo C monitor framework consists of about 14,000 lines of code, developed for Sun Sparc workstations running Solaris and the GNU C compiler. Most of the framework employs user-level techniques that are applicable to any robust C compiler. However, the code loader is specific to the ELF object format, the target program access library depends on stabs sections in GNU C format, and the memory protection facilities are provided by a UNIX mprotect() system call. The system was ported to x86-based Linux in less than a week, and would be readily ported to other ELF-based variants of UNIX.

2 Event-driven execution

Alamo is event-driven, as illustrated in Figure 2. Control switches back and forth between the execution monitors and the target program, transmitting *event requests* and replies in the form of *event reports*. Events are individual units of program behavior. Examples of typical events include program control flow, memory references, heap allocations, procedure calls and returns, clock ticks, and I/O operations. An event includes an integer code describing what is taking place, and a related target program value. The target program must be instrumented in order to produce events; this is a fundamental difference between Alamo and some kinds of monitors, such as traditional source-level debuggers.

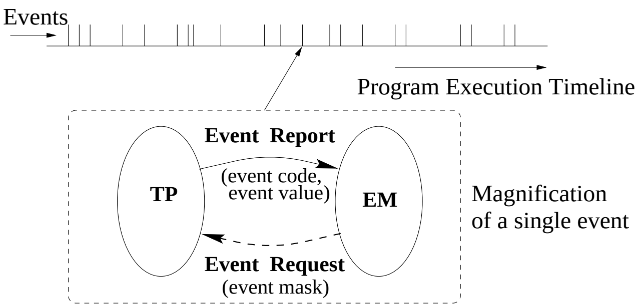

Figure 2: Event-driven execution.

Instrumentation

Instrumenting the target program by hand is not practical for large programs. Alamo employs automatic program instrumentation to produce target program events for monitors. The goal of Alamo's automatic instrumentation is to provide information at the semantic level of the source program, rather than the machine level. Providing monitors with higher-level information is one way to simplify monitor development.

Alamo's automatic instrumentation is application independent and comprehensive. The kinds of events available are driven by the target language syntax and semantics, including the semantics of its runtime libraries, rather than by any particular target program or monitor. This allows generic monitors to be written to observe the behavior of arbitrary programs. Although higher-level information is available for a variety of execution behaviors, monitors can obtain details when needed, down to the target-language basic blocks, memory references, and individual operators. The instrumentation may be categorized into two kinds: (1) basis events, derived from the C grammar, are pre-defined and describe behavior that is directly observable from the syntax; (2) configured events, derived from the configuration file, represent combinations or special cases of basis events that are instrumented to report higher-level behavior.

Automatic instrumentation can be accomplished by instrumenting the runtime system including library calls, or by inserting code directly into the source program. Instrumenting the runtime system is the simplest means of supplying monitors with events. It was suitable for the Icon framework; Icon programs spend the majority of their time executing runtime system code and the events produced from the runtime system have high-level semantic content due to that language's built-in control and data structures, algorithms, and memory management facilities.

The extent to which the runtime system behavior forms an adequate abstraction of overall program behavior depends on the language level, as well as on the program itself. Instrumenting the C runtime libraries might characterize some aspects of some programs' behavior adequately, but does not provide a general solution. In C, the behavior of interest often resides in the generated code. In the Alamo C framework, instrumentation of the target program is performed prior to compilation by a framework component called CCI, a Configurable C Instrumentation tool [5]. CCI is a preprocessor that generates instrumented C output. It includes a complete ANSI C compiler front-end and performs selective instrumentation by parse tree transformation.

Configuration

The main problem with an automatic C instrumentation tool is code blow-up, and the best solution is to perform static analysis comparable to those used in compiler optimization. In the case of CCI, unoptimized instrumentation of all available events results in object code that is about 50 times the size of the uninstrumented code. This size increase is due to in-lining of event filters to avoid context switches, described below. In any case, code blow-up is the reason CCI has a full compiler front-end, and the reason for its configuration mechanism.

In CCI, the granularity and semantic level of instrumented events are bounded at the low-end by the source-level C expressions that CCI is able to instrument based purely on syntax, and at the high-end by one or more *configuration files*. Configuration directives tell CCI what events to instrument; this compile-time method of event selection is complemented by dynamic event masking, discussed below.

More importantly, configuration directives also provide semantic information about runtime libraries and the application domain. Semantic information in turn allows for higher-level events, often composed from sequences of lower-level events. Where it is possible, it is important for performance to analyze the program and compose higher-level events at compile time rather than in the monitor at run-time. Configuration directives for standard libraries may be written once and shared by all target programs and subsequent monitors that use a given library. Application-specific configuration directives are specified by the end user in order to support application-specific monitors. The need to provide higher-level semantically-based information motivates the use of a compiler-style preprocessor for instrumentation instead of an object-file instrumentor.

An example CCI configuration file illustrates the declarative nature of CCI's configuration language. In this example, assignments to structures (E_Assigns) are instrumented, but only for structures foo and bar, and procedure calls are instrumented, but only to calls related to the heap. The built-in procedure call event E_Pcall is mapped into new event codes; malloc and calloc are instrumented to produce E_Alc events for basic heap allocation, and realloc is instrumented to produce E_Realc events for heap reallocation. Additional details are described in [5].

```
E_Assigns{struct foo, struct bar}
E_Pcall{malloc=E_Alc,calloc=E_Alc,realloc=E_Realc}
```

Filtering and masking

Automatic comprehensive instrumentation has a negative side-effect: even after configuration, the number of events produced may be far greater than the number actually needed by a given monitor. Conventional filtering methods in which monitors explicitly discard the unwanted events they receive provide inadequate performance in Alamo because each event report involves two lightweight context switches. These context switches are avoided by discarding unwanted events in the instrumentation code executed by the target program.

Each event request specifies the kinds of events desired with a set of integer event codes called an *event mask*. After a request, the target program's execution proceeds until an event occurs that is a member of the requested set. The event mask may be changed by a given monitor each time it requests an event, allowing it to narrow or broaden its set of desired events as needed. Event code masking is the primary event selection mechanism in both the C and Icon frameworks; event masks are efficiently implemented using bit vectors. Performance requirements for Alamo's C framework motivated additional selective power in the form of *value masks*, a separate set of values of interest may be supplied for each event code. For example, procedure call events could be restricted to a specific group of procedures by configuration at compile-time, but when more flexibility is called for the restriction may be dynamically imposed using a value mask at runtime. Figure 3 shows an event mask and a value mask associated with one of the event codes. Value masks are typically implemented using hash tables on addresses or values of interest.

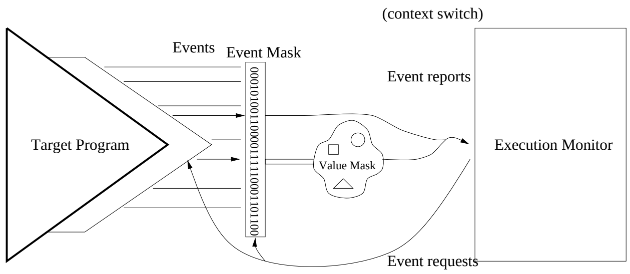

Figure 3: The event mask and value masks reduce the number of event reports.

3 Execution model

Traditional debuggers utilize a separate process, but two-process models of monitoring do not offer the inexpensive control mechanisms required to individually process "billions and billions" of units of target program behavior, nor the direct access to a program's memory regions that is needed to do analysis beyond what is reported by the events. It is in contrast to the classical two-process debugging model that Alamo's architecture can be considered lightweight.

Alamo provides an execution model in which a target program (TP) and the execution monitors (EMs) that observe it are *coroutines* executing within a single address space. A coroutine is a synchronous thread; in a coroutine execution model scheduling is non-preemptive and context switches are explicit [3]. Context switches within a single address space are lightweight, but some monitoring systems discussed in the Related Work section below offer an even less expensive alternative, which is to write the monitor code as a set of callback procedures.

Alamo executes monitors as coroutines instead of callback procedures in order to make monitors easier to write. Using a separate thread gives monitors their own "main" procedure and locus of control, and synchronous execution ensures that the program being monitored does not change state out from under the monitor while it is being examined. The goal of the model is to make monitor writing no more difficult than applications programming.

For trivial monitors that just count events or write them to a logfile, callback procedures are more suitable and may execute one to two orders of magnitude faster than an Alamo monitor. Alamo wasn't designed for event counters, but for monitors that perform more complex dynamic analysis tasks while the program is running, often with accompanying visualizations. For such monitors, the cost of the execution model is modest and the ease of programming afforded by the coroutine model enables more complex tasks to be attempted.

The coroutine execution model employed in Alamo was implemented identically for the C and Icon frameworks. Alamo's user-level coroutine switch is a small piece of assembler code that has been ported to many processors and operating systems. It was borrowed from existing code in Icon's implementation [4].

Monitor coordination

Alamo supports the development of multiple, specialized, user-level monitors that operate independently and may be mixed and matched as needed. The value of this type of microkernel architecture has been established in operating systems such as Mach and Windows NT. When multiple monitors are present in Alamo, their control and access to the target program is facilitated by a special execution monitor called a monitor coordinator (MC).

A monitor coordinator takes event requests from all monitors and activates the target program with the event mask union of those requests. The coordinator forwards each event report to those monitors that requested that type of event, providing them with the illusion that they are directly monitoring the target program themselves. Figure 4 illustrates typical monitor coordinator scenarios: (a) no coordinator, (b) coordinator forwarding events to a single monitor, and (c) coordination of multiple monitors. Since monitor coordinators are Alamo monitors, their implementation is itself open to experimentation; at present our coordinators are simple and are tuned to optimize common-case performance.

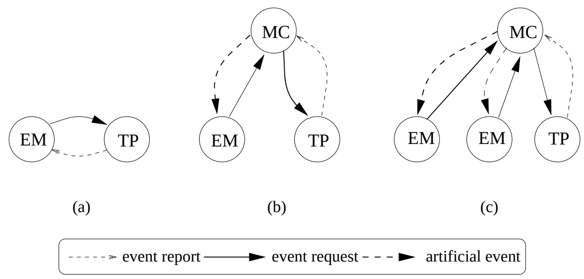

Figure 4: Monitor coordinator scenarios.

4 AME

The Alamo Monitor Executive (AME) forms the core component of the C framework. It loads relocatable ELF objects corresponding to the target program and the execution monitors, providing each program with the illusion that they are running in their own execution environment. Figure 5 shows the relationship between the AME and CCI within the Alamo C framework.

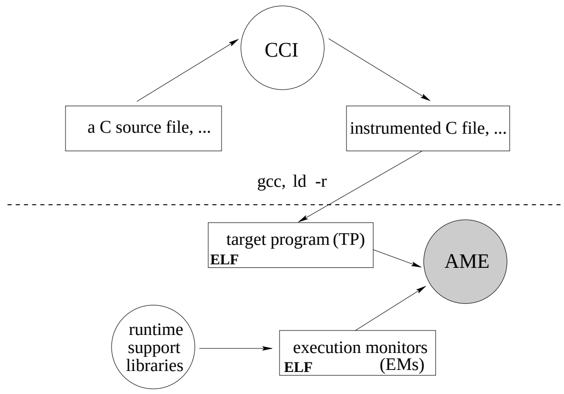

Figure 5: The Alamo C Framework.

Under AME, each program is loaded into its own data area with code and global data sections. Unlike most dynamic loaders, AME loads ELF objects without linking their symbols. In addition, employing a custom ELF code loader (based on code from the Linux kernel) allows the Alamo system to provide comprehensive access to the target program.

Heaps

In the C framework, loaded programs are provided with their own heap by means of a modified version of the GNU malloc() library. The modified library uses a fixed memory region for each target program and monitor, determined when they are loaded. This load-time limit on the heap size represents a limit of the C framework, although heap sizes may be set arbitrarily large. The limit could be removed for C programs that do not depend on a true sbrk() for contiguous heap expansion. The C framework also supports a mode in which monitors and the target program share the standard C heap, without limits but also without the separation that the independent heaps approach gives. Figure 6 shows the runtime environment provided by AME.

Memory protection

Memory protection in the Icon framework is implicit; the language has no pointers and is strongly typed at runtime, so a target program can perform no operation that violates monitor integrity. In the C framework, errant programs pose a real threat to monitors. Under Solaris and many flavors of UNIX, monitors can be protected from errant target programs using the mprotect() system call. Turning on and off memory protection each time the target program is entered and exited significantly degrades performance since it requires operating system intervention at every event request and report. The performance cost is proportional to the

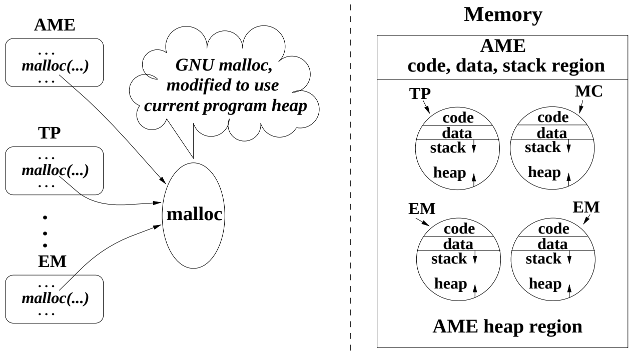

Figure 6: The AME run-time environment.

number of events reported; it will be highest for profilers and other monitors that request many events but do little with them.

Because of the performance degradation entailed and the fact that many monitors are written to work on programs that do not contain memory violations, memory protection is optional in the C framework. When a monitor requests the memory violation event, E_MemViol, the AME performs the necessary system calls to enable protection when the target program commences executing and then disable protection when an event report causes execution to switch back to the monitor. This allows a monitor to leave memory protection turned off in those portions of the target program where it is considered well-behaved, and then turn on memory protection when it reaches a stage in which the target program's memory references are not reliable.

5 Target program access

In addition to the information in events themselves, a monitor may inspect target program state using a library of access functions. Access functions extend the capabilities of the Alamo architecture beyond the event-driven paradigm and give it monitoring capabilities closer to those provided by a programmable debugger. Monitor writers use access functions for purposes such as obtaining names of target program variables, and manipulating target program values.

Type information is a key component of target program access for performing operations such as pointer arithmetic and structure field references. In the Icon framework, manipulating the target program's values or traversing its structures was trivial because type information is available at runtime and the language has a fixed set of structure types for a monitor to deal with.

In the C framework, the type information required for target program access is available only if the target program is compiled with the -g debugging symbol option enabled. Debugging symbol information is available in *stabs* sections. A model that bundles target program values with type information into *descriptors* simplifies the access functions and the use of this stabs information. [6] is a complete description of the C framework's target program access library.

6 Example monitors

Some example execution monitors below illustrate how Alamo's features are used to perform typical tasks in the C framework. All Alamo monitors are written starting from the following template. The monitoring system is initialized and then the monitor executes the target program in increments controlled by the event reporting mechanism until execution terminates, upon which the monitor may clean up and generate summary reports. The Alamo library routines used are EvInit(), EvGet(), EvTerm().

```
#include <alamo.h>
void main(int argc, char **argv)
{
    /* Initialize execution monitoring */
    EvInit(argc, argv);
    /* Sets up the initial eventmask for the EM */
    mask = EvMask(n, eventcode1,... eventcoden);
    while ( eventcode = EvGet(mask) ) {
        /* Process events */
        }
    /* Termination code */
    EvTerm();
}
```

This template is omitted from the following examples. In each example, replace the comment "process events" in the template with the switch statement in the example.

Checking array bounds

A common C bug is accessing an array element which is out of bounds. A monitor that detects this problem is given below. This example demonstrates the capabilities of the target program access functions that enable a monitor to inspect additional state information when it processes an event.

The events related to array access include array referencing (E_Array) and indexing (E_Index). The event value for an array reference event is the memory location of the array referenced. Variable name and type information is obtained from the EvStab() access function. The event value for an array index event is the integer subscript used. This example uses a stack of structures that consist of a character pointer to store the array name and a descriptor to store its memory location and type information. The stack is required because several array reference events may occur before the corresponding index events are resolved in the case of array references within array subscripts, such as a[a[i]].

```
switch (eventcode) {
    case E_Array:
        EvStab(eventvalue, &(array_stack[level++]));
```

```
            break;
        case E_Index:
            elem = EvElem(array_stack[−−level].desc,
                    eventvalue);
            if (IsNull(elem))
                fprintf(stderr, "index out of bounds:%s[%d]\n",
                    array_stack[level].name, eventvalue);
            break;
}
```

Visualizing Tree Structures

The above example shows that easy forms of monitoring are easy in Alamo; visualizing more interesting dynamic behavior, such as structural changes and access patterns within a program's tree structures illustrates the kind of execution monitor Alamo was really designed to support. One such monitor is CTV, a C Tree Visualizer [14]. CTV is implemented in about 1200 lines of code.

CTV is an Alamo monitor that extracts and visualizes tree behavior within C programs. It processes events, looks for references to values whose types are recursive, and visualizes the trees it finds in the target program. CTV converts sequences of low level events such as memory references into higher level *tree-manipulation events* using a pushdown automaton shown in Figure 7. A sequence of events starting with an E_Refp and ending with an E_Assignp is converted into creation of a new tree node (top cycle) or insertion into an existing node (bottom cycle).

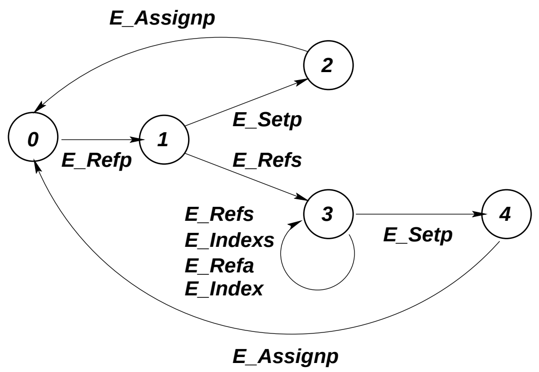

Figure 7: Constructing tree events from lower-level events.

The higher-level the information provided by instrumentation, the less work will be required of the monitor writer. In the case of trees, the work of the pushdown automaton—detecting appropriate structure types and accesses—can be moved to compile time by a sufficiently powerful automatic instrumentation tool, which will make tools like CTV easier to implement.

The resulting trees detected by CTV are visualized using an OpenGL rendering inspired by molecular models. A sample tree of depth 8 is shown in Figure 8. More sophisticated tree layout algorithms are available that scale better to very large trees, such as cone trees [15] or hyperbolic trees [16].

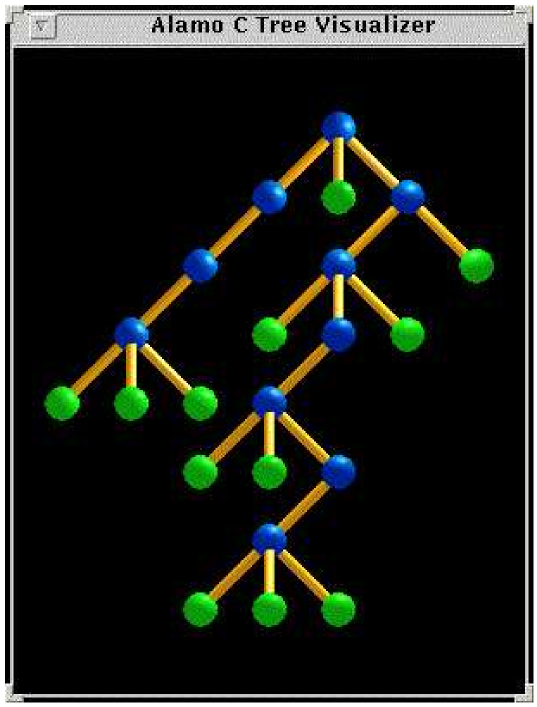

Figure 8: A tree constructed with the tree visualizer.

7 Performance

Preliminary performance results for the Alamo C framework are given in Table 1. The primary performance issues are (1) the cost of the instrumentation, consisting of assignments to temporary variables for event values and tests of whether to report an event; (2) the cost of the context switches between monitors and target program when events are reported, consisting of register saves and restores; and (3) the cost of protecting monitors from errant target programs, an operating system call. Overall performance depends on the extent of the analysis performed by the monitors. The cost of the framework depends on how well the monitors are able to focus the reported behavior by means of static configuration and dynamic masking.

The target programs are laplace.c, and life.c. *laplace.c* approximates the solution to Laplace's equation at a point in an annulus via a Monte Carlo method. The solution is approximated using 1000 random walks. *life.c* is Conway's Game of Life; executed for 30 generations.

	-O	-g	event	coswitch	mprotect
laplace.c	0.62	0.68	1.42	137.54	2130.15
life.c	0.22	0.26	1.52	143.66	2160.37

Table 1: Execution time (in sec) for target programs in five different cases.

The third column of numbers shows the cost of Alamo's instrumentation and motivates selective (configurable) automatic instrumentation techniques; the times given are worst-case scenarios where every event is instrumented. In that case, the presence of tests to decide whether to report events may impose a slowdown factor of 2-6 even if no events are reported. The fourth column shows the cost of Alamo's event reporting mechanism based on lightweight context switches. Two orders of magnitude in execution speed are lost in these worst-case scenarios where every possible event is reported to a monitor; this motivates event masking, which reduces the number of events that are actually reported. The fifth column shows the cost of invoking UNIX memory protection features to protect the execution monitors from the target program. Over an order of magnitude performance penalty is incurred by this feature, when it is needed.

8 Related work

Alamo can be compared to several existing systems. EEL and OM are object file instrumentation tools that instrument behavior at the instruction level [13] [17]. Object-code modification allows monitoring of programs even when source code is unavailable, and allows monitoring of behavior that Alamo does not even consider, such as register usage or instruction scheduling, but it is oriented towards building monitors that describe architectural behavior rather than application behavior. The ATOM system, built on top of OM, is a framework provides support for building instruction-level monitors similar to the support that Alamo provides for building higher-level tools [18].

The key contrasts between systems such as ATOM and Alamo are: the notation available and semantic level of the instrumentation, the performance of the event delivery mechanism, and the programming model provided to monitor writers. In ATOM, the monitor writer specifies instrumentation by writing a program that works with abstractions such as basic blocks and machine instructions. In Alamo, the monitor writer specifies instrumentation in a declarative set-based notation, works with abstractions such as the program's defined types and interfaces, and can re-use instrumentation for standard libraries to obtain higher-level events for such functions at no cost. The performance of ATOM is close to optimal, and for comparable events ATOM's procedure-call execution model should run an order of magnitude faster than Alamo's coroutine model. Alamo monitors tend to work with fewer, higher-level events than ATOM monitors.

The contrast of real note is the programming model. Using ATOM, monitors are written as a series of callback procedures. Any sort of state maintained between events must be stored in global or static variables. Callback-based programming tends to consist of procedures with lots of large switch statements; switches on the event type, switches on the values being observed, and switches on the implicit state that is maintained between events. In Alamo monitor writing is still a complex task, but monitors have their own main() procedure, their own locus of control, and their state is maintained in between events, which they obtain by performing an ordinary-looking function call, rather than by being called.

Several other existing monitoring systems are important and form interesting contrasts with Alamo. IBM's PV system provides visualization tools with events for program behavior at multiple levels of abstraction, including operating system and hardware levels not considered here [8]. PV provides automatic instrumentation of lower-level behavior, but higher-level events are hand-instrumented. The UW Illustrating Compiler UWPI is a system that infers higher-level semantic information in the form of abstract data types for variables in programs written in a subset of Pascal; the abstract data types are then used to produce visualizations of data structures [7]. UWPI is not an architecture for constructing monitors, but an instance of a monitor that exploits compiler information to improve information in a manner similar to Alamo.

Dalek is a flexible programmable debugging system based on gdb; it provides an execution model as convenient as Alamo's, but suffers in the area of performance. Its authors point out performance limitations for monitoring using the conventional two-process execution models employed by source-level debuggers and many monitoring frameworks[11]. Dynascope is a system that employs interprocess communication to allow a program to direct the execution of several programs, possibly on different machines [9] [10]. Because the directing server is a separate process it has its own locus of control, Dynascope must deal with similar performance and communication issues as Dalek. But Dynascope uses a hybrid model in which a monitoring function library is embedded into the program being monitored. Dynascope is geared towards monitoring larger-grained behavior such as distributed communication for which it is suited.

BEE++ is a C++ monitoring framework in which programs are instrumented by hand or by subclassing instrumented classes, and specialized monitors can be written by subclassing more general monitors [12]. Although hand-instrumentation is an unattractive prospect, instrumented versions of standard class libraries would allow high-level semantic information to be obtained without effort for C++ applications that use those class libraries.

9 Conclusions

The Alamo monitor architecture significantly reduces the development cost of writing program execution monitors. The design has been realized by monitor frameworks for two very different programming language implementations. The C framework that has been developed required a substantial systems programming effort, which can now be avoided by programmers engaged in exploratory development of new kinds of monitors such as program visualization tools for C programs. Monitor performance under Alamo is quite attractive when the available static and dynamic means of reducing the number of reported events are employed.

The Alamo architecture has inherent limitations. There is no support for real-time or shared-memory multiprocessor-based parallel applications. Not all execution monitors can be written using an Alamo-based framework; those that cannot tolerate intrusion of instrumentation code require a two-process model such as that employed by standard source-level debuggers.

10 Acknowledgements

Ralph Griswold contributed to the development of the Icon framework from which Alamo sprang. Robert Shenk and Sandra Dykes read this manuscript and made numerous useful suggestions.

References

[1] Bernhard Plattner and Jurg Nievergelt, "Monitoring Program Execution: A Survey," IEEE Computer, November 1981, pp. 76-93.

[2] Clinton L. Jeffery and Ralph E. Griswold, "A Framework for Execution Monitoring in Icon," Software—Practice and Experience, Vol. 24(11), November 1994, pp. 1025-1049.

[3] C.D. Marlin, "Coroutines (Lecture Notes in Computer Science 95)", Springer-Verlag, Berlin, 1980.

[4] Steven B. Wampler, "The Control Mechanisms for Generators in Icon", University of Arizona Department of Computer Science TR 81-18, December 1981.

[5] Kevin S. Templer and Clinton L. Jeffery "The Design of a Configurable C Instrumentation Tool", University of Texas at San Antonio Division of Computer Science TR 96-6, February 1996.

[6] Wenyi Zhou and Clinton L. Jeffery, "Target Program State Access in the Alamo Monitor Framework", University of Texas at San Antonio Division of Computer Science TR 96-5, February 1996.

[7] Robert R. Henry and Kenneth M. Whaley and Bruce Forstall, "The University of Washington Illustrating Compiler", in Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, pp. 223-233.

[8] Doug Kimelman and Bryan Rosenburg and Tova Roth, "Strata-Various: Multi-Layer Visualization of Dynamics in Software System Behavior", in Proceedings of IEEE Visualization '94.

[9] R. Sosic, The Dynascope Directing Server: Design and Implementation, *USENIX Association, Computing Systems*, Vol. **8**(2), 107-133, 1995.

[10] R. Sosic, A Procedural Interface for Program Directing, *Software Practice and Experience*, Vol. **25**(7), 767-787, July 1995.

[11] Ronald A. Olsson and Richard H. Crawford and W. Wilson Ho, "Dalek: A GNU, Improved Programmable Debugger", in Proceedings of the USENIX Summer '90 Conference, June 1990, pp. 221-231.

[12] B. Bruegge, T. Gottschalk, and B. Luo, A Framework for Dynamic Program Analyzers, *OOPSLA '93 Proceedings, Sigplan Notices*, 28(10), 65-82, Oct. 1993.

[13] J. R. Larus and E. Schnarr, EEL: Machine-Independent Executable Editing, *SIGPLAN Conference on Programming Language Design and Implementation*, June, 1995.

[14] Michael Chase Brazell and Clinton L. Jeffery, Tree Structure Detection and Visualization, Technical Report CS-97-7, Division of Computer Science, University of Texas at San Antonio, August, 1997. www.cs.utsa.edu/research/alamo/tr97_7/

[15] George G. Robertson, Jock D. MacKinlay, and Stuart K. Card, Cone Trees: Animated 3D Visualizations of Hierarchical Information, Proceedings of CHI '91, New Orleans, April 1991, pp. 189-194.

[16] John Lamping and Ramana Rao, Layout out and Visualizing Large Trees Using a Hyperbolic Space, Proceedings of UIST '94, Marina Del Rey, November 1994, pp. 13-14.

[17] A. Srivastava and D. W. Wall, A Practical System for Intermodule Code Optimization at Link-Time, *Journal of Programming Language*, **1**(1), 1-18, March 1993.

[18] A. Srivastava and A. Eustace, ATOM: A System for Building Customized Program Analysis Tools, *Proceedings of SIGPLAN '94 Conference on Programming Language Design and Implementation*, 196-205, ACM, 1994.