Electronic Notes in Theoretical Computer Science 30 No. 4 (2000)
URL: http://www.elsevier.nl/locate/entcs/volume30.html 18 pages

The Alamo Execution Monitor Architecture

Clinton L. Jeffery [1]

Department of Computer Science
University of Nevada, Las Vegas
NV, U.S.A.

Abstract

Future programming environments will incorporate a tighter coupling between language runtime systems and the monitoring tools that are used to debug, tune, visualize, and understand them. Many innovations that are developed first in higher level programming language environments will migrate into mainstream languages once their properties are understood and generalized.

The Alamo execution monitor architecture was developed to facilitate rapid development of execution monitors, especially visualization tools that are instrumental in understanding complex runtime system interactions in higher level languages. Alamo simplifies the development of such tools by solving the low-level access, control, and intrusion problems inherent in monitoring.

Alamo was implemented first for the very high-level imperative goal-directed language Icon. The architecture was then implemented for ANSI C in order to broaden the impact of the work. This paper describes the ANSI C implementation of Alamo and the monitoring services it provides.

1 Introduction

Alamo is a software architecture that reduces the costs of developing execution monitors. The name *Alamo* stands for A Lightweight Architecture for MOnitoring. Alamo is described extensively in [1]. Alamo's primary domain is that of visualization tools used in debugging and program understanding. Such monitors present a visual analysis of the dynamic behavior of programs.

Alamo was created by generalizing monitoring facilities that were developed for Icon, an interpreted language [2]. After using these facilities in classes

[1] This work supported in part by the National Science Foundation under grants CCR-9409082 and CDA-9633299

and writing a large number of visualization tools, a similar monitoring framework for ANSI C was created. The Alamo architecture captures those principles relevant to monitors for both compiled and interpreted languages, as opposed to the implementation details that vary from language to language.

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.

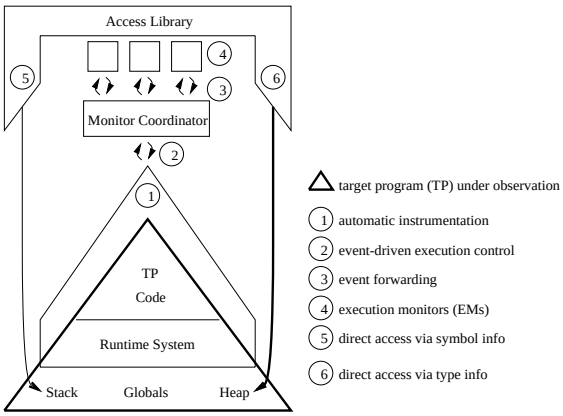

Fig. 1. The Alamo architecture.

This paper describes aspects of Alamo's C implementation. It summarizes and extends the work presented in [3] and [4]. 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.

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.

Alamo is event-driven. 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.

2 Instrumenting ANSI C

Software instrumentation systems must solve several problems, such as inserting instrumentation code while preserving program semantics, minimizing code blowup, and reducing the execution slowdown due to the huge number of events. Tools that analyze and depict behavior at the same time the program is running must also provide efficient delivery of events to the monitor. Extracting high level behavior from low level events is a problem for either the instrumentation system or the execution monitor. The more support for this task that the instrumentation system provides, the easier it is to write monitors.

CCI is Alamo's Configurable C Instrumentation tool. CCI inserts events into the target program's source code. In CCI an *event* represents any unit of program behavior that is observable at the source level of the C language, such as variable references, function calls, or control flow. While these events are not much higher level than those in equivalent object code level instrumentation, CCI's configurability provides a viable means of extracting high level events that is difficult to duplicate at the object code level.

2.1 CCI Design Overview

CCI parses the desired source code files, performs semantic analysis, inserts instrumentation based on one or more configuration files, and produces new files containing instrumented source code. CCI then calls an ANSI C compiler such as gcc to compile the instrumented source code.

CCI inserts an event macro `EV()` that client monitors define as needed. With different macro definitions, monitoring systems can produce log files, send information via a network connection, or directly execute monitor code via a function call. The Alamo framework defines CCI's event macro to perform dynamic filtering and lightweight context switches to the monitors, which

execute as coroutines of the program being monitored[3].

Instrumented code suffers runs slower than uninstrumented code by a factor that depends on the execution model of the client monitor and its event macro. CCI's configuration mechanism reduces the cost of instrumentation by allowing a monitor to tailor the instrumentation to its needs.

2.2 Instrumentation

CCI's available instrumentation consists of a predefined set of *basis events* and a configuration mechanism for selecting, refining, and composing these basis events into higher level events. The basis set is comprehensive and represents fairly low level behavior directly observable from the syntax; it is derived from the C grammar. The complete set of basis events is given in [5].

CCI's event macro EV() takes *event code*, and *event value* parameters. Event codes are integers that describe the kind of behavior that has occurred. Event values are target program values, or pointers to values, whose types vary with the event code; each kind of event has a corresponding value type. For example, E_Addi is an event code for integer addition; its event value is of type integer.

CCI instruments by inserting event nodes, *enodes*, into a parse tree. Figure 1 shows an example of the parse tree that CCI constructs for the expression a = b + c, with and without instrumentation.

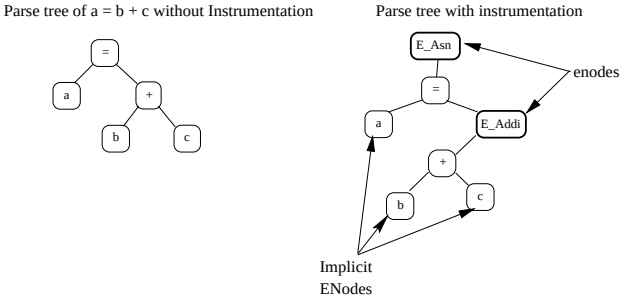

Fig. 2. Example of CCI's parse tree construction

2.3 Type information and Symbol Tables

In addition to parse tree construction, CCI maintains symbol table information similar to any compiler. Many of the events that CCI reports have type information embedded in their event code. For example, the event code for an integer addition, E_Addi, differs from float addition, E_Addf. As described earlier, CCI uses this type information to help distinguish events. CCI uses a mnemonic to make event codes easier to remember and understand.

For polymorphic operators with several value types, multiple event codes are used and the event code name is augmented with type specifiers. For example E_Addi, E_Addf, E_Addd, E_Addld, E_Addp, and E_Addc correspond to addition events of integers, floats, doubles, long doubles, pointers, and characters respectively. CCI does not use event codes to delineate storage class, type qualifiers (e.g. const and volatile), or signed and unsigned.

2.4 Instrumented Code

An example of CCI's output for the statement a=b+c; is shown below. In this example, all events in the basis set are instrumented.

```
(EV(E_Seti,_Tcci_1=EV(E_Refi,&a)),
    EV(E_Assigni,*_Tcci_1=((_Tcci_0=(
        (EV(E_Refi,&b),(b))+
        (EV(E_Refi,&c),(c))),
    EV(E_Addi,_Tcci_0),_Tcci_0))),*_Tcci_2);
```

CCI creates temporary variables to hold intermediate results of instrumented expressions, making extensive use of the comma operator to preserve the original expression semantics. Even for a simple expression such as a = b + c, there is a problem of code explosion. To reduce this problem, CCI produces instrumentation only for behavior specified in a configuration file. For example, if the configuration were set to only instrument E_Add events, the above instrumentation would reduce to:

```
a = ((_Tcci_0 = b + c),EV(E_Addi,_Tcci_0),_Tcci_0);
```

3 Configuration

Without configuration, CCI produces huge amounts of code intrusion, with a significant performance penalty. In order to provide flexibility and support a wide range of monitors, CCI makes available low level events which require a fair amount of work to interpret and use. Processing these events at run time would add a significant additional cost to monitoring.

Configuring CCI to produce fewer, higher level events significantly reduces code intrusion and moves many run time computations to compile time. The monitor writer configures CCI to specify the events that are needed and how to present them. Moreover, the monitor writer can apply knowledge of the program's source code or the run time libraries it uses to guide CCI in generating more meaningful high level events. Configuration is useful for application specific monitors, but configuration information for standard headers and libraries raises the semantic level of events for all applications that use those

modules.

CCI's configuration language has facilities for selecting, refining, and composing basis events into higher level events. The grammar for CCI's configuration language is given in [5]. In order to perform these operations, the monitor writer must know and understand the primitive events that are defined in the basis set, which are rooted in C syntax and semantics. The declarative nature of the configuration language makes it easy to manipulate and use without programming. A configuration file contains one or more event set *selections* and/or *definitions*, defined later.

CCI applies instrumentation based on scoping rules that are similar to C *global scope* and *local scope*. Global event selections appear at the beginning of the configuration file and apply instrumentation to the entire program. Local scope event selections apply instrumentation to a particular function or range of line numbers. If multiple configuration files are used, CCI merges the global selections of each file into one global selection and merges the local selections as if all configuration information were contained in one large file.

3.1 Event Selection

As described above, CCI has a basis set of approximately 160 event codes that represent the lowest level units of program behavior. The basis set defines events for behavior such as assignment, variable dereferences, procedure calls, control flow, array index, and structure accesses. During configuration, monitor writers select desired events from CCI's basis set.

An event set selection is a list of one or more event names. If an event set is selected in the global section of the configuration file, CCI applies the selection globally and will generate events for all the source files that contain those events.

If the user wants to instrument a specific function, the user types the function name in brackets and lists any desired event selections. For example, to instrument the function heap_sort one might write:

```
[heap_sort]
E_Refa, E_Assign, E_Pcall, E_Pret;
```

In C there may be many (static) functions with the same name but in different source files. CCI uses a filename.function syntax in such cases.

3.2 Filtering and Refinement

CCI's instrumentation provides static filtering based on types of events, which often boils down to the type of value being manipulated. An important feature of CCI's static filtering is that it may *refine* the events. Refining is the process

of taking one general event and narrowing it to a more specific event. Using the above examples as motivation, say that the monitor writer is creating many structure masks and writes:

E_Assigns{struct foo, struct boo, struct coo}

When an assignment to *any* of these structures occur, CCI will generate the general event E_Assigns. However, through refining more specific information is extrapolated:

```
E_Assigns{struct foo=E_AssignFoo,
          struct boo=E_AssignBoo,
          struct coo=E_AssignCoo}
```

Here, a new, monitor-writer-defined event name is generated for assignments for each of these structures.

The value of static filtering in configuration is considerable. Type information enables CCI to generate events at a higher level than the machine instructions, and type information can be employed at instrumentation time to reduce run-time costs.

4 The Alamo Monitor Executive

Instrumented programs are loaded and run along with monitors by a controlling program called ame, for Alamo Monitor Executive. AME must provide inexpensive control mechanisms required by monitors that individually process "billions and billions" of units of target program behavior, as well as 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.

4.1 Execution Model

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 [6]. 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 [7].

4.2 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 3 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.

4.3 Dynamic loading

AME 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 4 shows the relationship

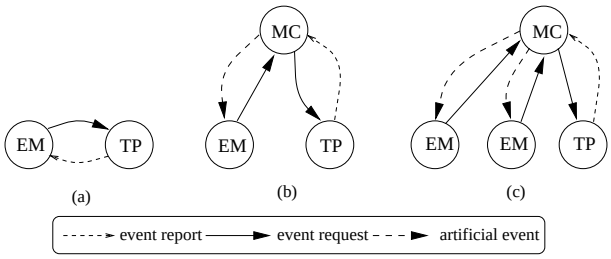

Fig. 3. Monitor coordinator scenarios.

between the AME and CCI within the Alamo C framework.

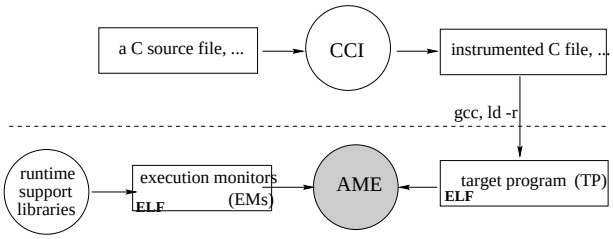

Fig. 4. The Alamo C Framework.

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.

4.4 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 5 shows the runtime environment provided by AME.

4.5 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 pro-

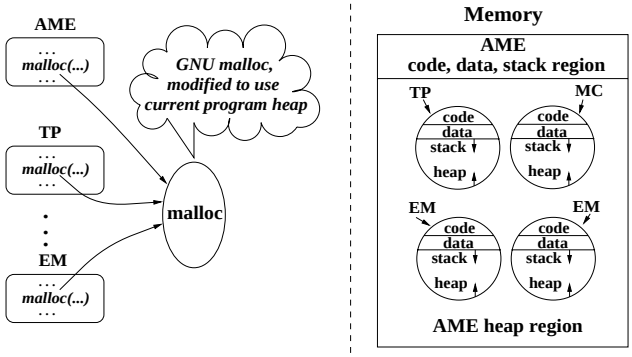

Fig. 5. The AME run-time environment.

grams 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 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.

4.6 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. [8] is a complete description of the C framework's target program access library.

5 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(), EvMask(), EvGet(), and 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, eventcode_1,... eventcode_n);
    while ( eventcode = EvGet(mask) ) {
        /* Process events */
        }
    /* Termination code */
    EvTerm();
}
```

This template is omitted from the following example. Replace the comment "process events" in the template with the switch statement in the example.

5.1 Checking array bounds

A common C bug is accessing an array element which is out of bounds. Event handling for an Alamo 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_Refa) 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_Refa:
        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;
}
```

5.2 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 [9]. 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 6. 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).

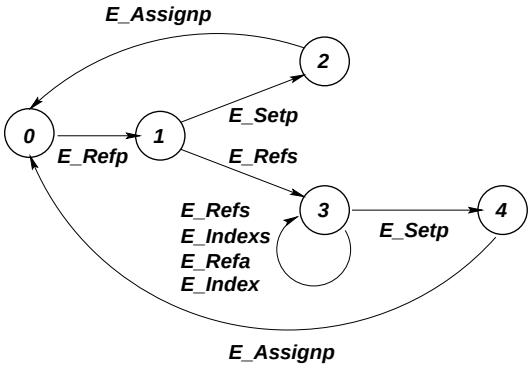

Fig. 6. Constructing tree events from lower-level events.

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 7. More sophisticated tree layout algorithms are available that scale better to very large trees, such as cone trees [10] or hyperbolic trees [11].

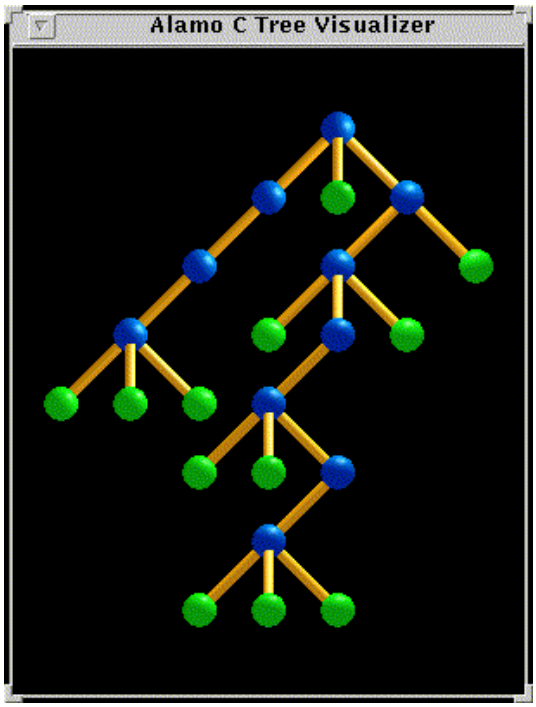

Fig. 7. A tree constructed with the tree visualizer.

6 Performance

A monitor framework is useless if its performance cannot meet the requirements of execution monitors. Separate performance observations measure intrinsic costs of the instrumentation method and the Alamo synchronous shared-address execution model.

The primary performance issues in the Alamo Executive 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.

6.1 Performance of Instrumented Code

CCI's comprehensive instrumentation is intrusive, both in code size and execution speed. CCI's configuration facility reduces the cost of automatic instrumentation to acceptable levels. While acceptable for the intended purpose of supporting tools such as program visualizers, CCI's instrumentation may not be suitable for very large programs or monitoring programs with real time performance constraints. Detailed results are presented in [5].

6.1.1 Code Explosion

Programs instrumented using CCI have code size increases on the order of *O(P * (I(P)+MC(P)))*, where P is program size, *C(P)* is the number of events in P selected by the configuration file, M is the size of the macro definition provided by the monitor, and I is the implicit cost of CCI's instrumentation method, primarily due to inserted temporary variable assignments. Although *I(P)* and *C(P)* vary from application to application, it is easy to think of them as constant coefficients, especially for very large programs, giving *O(P * (I+MC))*. Although this is *O(P)*, the *(I+MC)* constants may be intractably large.

Code explosion factors typically range from 3-5 when tiny macros and narrow event selections are used; these terms were dominated by the cost of CCI's instrumentation method and show the range of I. Further optimizations to CCI or supplied by the underlying C compiler may reduce I to a negligible amount, although costs of temporary variables will still be incurred proportional to C.

On the other hand, a large event macro results in code explosion factors ranging from 10 to 60 depending upon the configuration. The larger the macro definition cost M, the more critical it becomes to provide higher level config-

urations that select narrow behaviors of interest and reduce the configuration factor C. M varied by close to an order of magnitude depending on whether a simple callback was used or a more substantial in line bit vector test was performed by the macro to determine whether to call the the monitor code.

6.1.2 Execution Time Cost

The execution slowdown equation imposed by CCI is similar to its space increase, except that M can be vastly larger for monitors that interact with the operating system, for example to do context switching. Execution slowdowns range from 2-4 for in-line counters, and from 4-9 for monitors that execute as a set of callback routines.

6.2 Performance of the Alamo Executive

Overall monitoring performance depends on the extent of the analysis performed by the monitors. The cost of the Alamo framework depends on how well the monitors are able to focus the reported behavior by means of static configuration and dynamic masking. This section summarizes results reported in [3].

The slowdown introduced by the coroutines in the Alamo C framework is huge compared with the intrinsic cost of instrumentation. The time cost the event macro M is gigantic, due to the runtime cost of a lightweight context switch compared with the typical C operators and control structures being observed. Configuring CCI to select specific types and operations can reduce the execution slowdown imposed from three orders of magnitude down to under one order of magnitude: as instrumentation becomes more selective, the configuration cost C becomes quite small. For example, while the game of life performs a total of 7.9M observable Alamo events, only 314K of those events are needed by the array reference monitor.

Alamo allows monitors to utilize UNIX memory protection features to protect themselves from the target program. Over an order of magnitude performance penalty is incurred by this feature, when it is used. These are again worst-case figures; the fewer actual event reports after configuration and event masking, the less overhead imposed by memory protection.

Monitors that do complex analysis and/or render complex visualizations using a toolkit such as OpenGL may impose more execution slowdown than the Alamo overhead, especially when configuration and masking are in use.

7 Related work

A large number of interesting related systems are described in [12] and in [1]. 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 [13]. PV provides automatic instrumentation of lower-level behavior, but higher-level events are hand-instrumented.

Dalek is a flexible programmable debugging system based on gdb; it provides an execution model as convenient as Alamo's, but suffers from performance limitations inherent in the two-process execution models employed by source-level debuggers and many monitoring frameworks[14]. The Dynascope system employs interprocess communication to allow a program to direct the execution of programs [15] [16]. Dynascope has some of the same performance and communication issues as Dalek, but uses a hybrid model in which a monitoring function library is embedded into the program being monitored.

Like CCI, Tolmach and Appel's debugger for Standard ML uses preprocessor-style source code instrumentation [17]. This allows debugging facilities to function properly in the presence of extensive optimizations such as those performed by the SML NJ compiler. Their instrumentation is tied to a specific debugger, and is always inserted at the fixed set of site types that it supports.

The Opium debugger for Prolog [18] offers several features advocated in the Alamo monitor architecture. Opium supports monitors written in a very high level language, advocates a coroutine execution model, and performs prefiltering of events in the target program similar to (and for the same reason as) Alamo's dynamic event masking.

8 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 acceptable when the available static and dynamic means of reducing the number of reported events are employed. CCI assists in this process by producing instrumentation that is both flexible and high level and has facilities for improving performance. CCI reduces code

explosion in instrumented code by providing configuration in which the monitor writer selects the events they are interested in. In addition, configuration moves filtering from run time into compile time, improving performance and simplifying the monitor writer's job.

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.

Alamo provides a usable exploratory programming environment for experimental development of new monitors. Production versions of Alamo monitors that prove to be useful may be re-coded for performance, for example to replace lightweight context switches on events with callback procedures. It would be useful to further develop Alamo's CCI instrumentation tool so that it supports hybrid instrumentation in which some events were handled by callback procedures and others result in context switches.

9 Acknowledgements

Wenyi Zhou and Kevin Templer implemented the Alamo C Framework in two splendid complementary Master's theses. Michael Brazell wrote the C Tree Visualizer program while completing his undergraduate degree.

References

[1] Jeffery, C., "Program Monitoring and Visualization," Springer, New York, N.Y., 1999

[2] Griswold, R., and M. Griswold, "The Icon Programming Language," Peer-to-Peer Communications, San Jose, 1997

[3] C. Jeffery, W. Zhou, K. S. Templer, and M. Brazell. A lightweight architecture for program execution monitoring. *ACM PASTE'98 Workshop, published in SIGPLAN Notices*, 33(7):67–74, July 1998.

[4] K. S. Templer and C. Jeffery. A configurable automatic instrumentation tool for ANSI C. *Proceedings of the IEEE 13th International Conference on Automated Software Engineering, ASE'98*, Honolulu Hawaii, October 1998.

[5] K. S. Templer. Implementation of a configurable C instrumentation tool. *Master's Thesis, Division of Computer Science, University of Texas at San Antonio (www.cs.utsa.edu/research/alamo/cci.ps.gz)*, May 1998.

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

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

[8] 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.

[9] 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/

[10] 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.

[11] 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.

[12] John Stasko, John Domingue, Marc Brown, and Blaine Price, editors. "Software Visualization: Programming as a Multimedia Experience," MIT Press, Cambridge, MA, 1998.

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

[14] Ronald A. Olsson, 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.

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

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

[17] A. Tolmach and A. Appel. A debugger for standard ML. *Journal of Functional Programming*, 5(2):155–200, April 1995.

[18] M. Ducassé. A general trace query mechanism based on Prolog. *International Symposium on Programming Language Implementation and Logic Programming, published in Springer-Verlag Lecture Notes in Computer Science*, (631):400–414, August 1992.