A Configurable Automatic Instrumentation Tool for ANSI C*

Kevin S. Templer and Clinton L. Jeffery
(ktempler|jeffery)@cs.utsa.edu
Division of Computer Science, University of Texas at San Antonio

Abstract

Automatic software instrumentation is usually done at the machine level or is targeted at specific program behavior for use with a particular monitoring application. This paper describes CCI, an automatic software instrumentation tool for ANSI C designed to serve a broad range of program execution monitors. CCI supports high level instrumentation for both application-specific behavior as well as standard libraries and data types. The event generation mechanism is defined by the execution monitor which uses CCI, providing flexibility for different monitors' execution models. Code explosion and the runtime cost of instrumentation are reduced by declarative configuration facilities that allow the monitor to select specific events to be instrumented. Higher level events can be defined by combining lower level events with information obtained from semantic analysis of the instrumented program.

1 Introduction

Program execution monitoring is an important tool for use in debugging, performance tuning, and program understanding. Many execution monitoring systems obtain information by instrumenting code at the machine level. This approach has been used to simulate and analyze architectural features such as cache systems, and to profile systems' fundamental performance characteristics, but machine level information is too low level for program visualization and debugging of high level program behavior.

The complexity of modern programs creates a demand for increasingly sophisticated monitoring tools[3], but developing program execution monitors is a difficult task requiring much effort to obtain and correctly interpret low level information obtained from the program being monitored. Furthermore, because many monitoring tools obtain information at the instruction level, monitor writers usually have little help in extracting the higher level behavior that is of greater value in debugging and visualization. Recurring difficulties that arise in software instrumentation systems include:

- location and technique used to insert instrumentation code
- code blowup
- execution slowdown due to the huge number of events

In tools that analyze behavior at run time (as opposed to post mortem systems), the efficient delivery of events to the monitor is also a primary concern. The extraction of higher 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, a Configurable C Instrumentation tool, instruments C programs for the purposes of monitoring and visualization. CCI instruments a target program by inserting 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. Example events include variable references, function calls, control flow, etc. 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.

CCI is a general purpose tool that was developed for Alamo, A Lightweight Architecture for Program Execution Monitoring[2]. The Alamo framework for ANSI C

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

uses CCI to obtain events from a target program which are then used in a variety of monitors, especially program visualization tools. CCI can easily be used by other monitoring applications.

In this paper we present the design of CCI and describe the configuration mechanism that enables CCI to instrument programs in terms of higher level concepts such as abstract data types instead of instruction level or syntactic source level information. CCI's configuration language allows the programmer to succinctly specify and obtain the behavior they are interested in and rapidly move on to debugging or visualization.

2 CCI Design Overview

CCI is a preprocessor that 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. Once CCI has instrumented the files, it calls an ANSI C compiler such as gcc to compile the instrumented source code.

CCI is event based. An event is an individual unit of program behavior[1]. CCI inserts an event macro that client monitors can define as needed for their particular execution model. With different macro definitions, monitoring systems might use the same instrumented program to produce log files for post mortem analysis, to send information to the monitor via a network connection, or to 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[2].

Because instrumenting code by definition involves code intrusion, instrumented code suffers from performance degradation 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.

2.1 Instrumentation

CCI constructs and maintains symbol tables and parse tree information in order to insert instrumentation code without altering the program's behavior. The 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 [7].

CCI's event macro `EV()` takes two parameters, an *event code*, and an *event value*. The uniform interface in which all events have one event code and one event value makes it simple and straightforward for monitors to interface with CCI. Event codes are integers that describe the kind of behavior that has occurred. Event values are target program 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. The type of event for `E_Refi`, which is a variable reference event, is a pointer to an integer because the value of the event is the address of the variable being referenced.

In order to correctly instrument syntactic levels of a source program, CCI builds a parse tree of the source. As the parse tree is constructed, CCI instruments the source by inserting event nodes, *enodes*, into the tree. Figure 2.1 shows an example of the parse tree that CCI constructs for the expression `a = b + c`, with and without instrumentation.

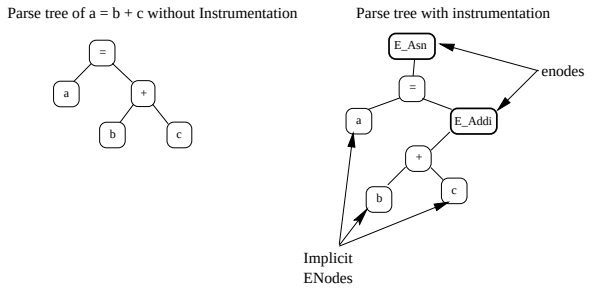

Figure 1: Example of CCI's parse tree construction

2.2 Enode Insertion

For every event in the basis set there is a direct mapping of that event to some rule or rules in the C grammar. An example YACC fragment showing CCI's parse tree construction for a function call with no parameters is shown in Figure 2:

```
postfix_expression:
  postfix_expression LP RP {
     $$ = ENode(E_Pcall,$1);
     }
```

Figure 2: Enode insertion CCI's yacc grammar

`ENode()` is a function which takes as its parameters an event code and a pointer to a parse tree node (`$1`). `ENode()` then wraps the parse tree node with another node that contains the event and returns the

newly wrapped subtree. If an event is not selected in the configuration file, `ENode()` returns the subtree that was passed in. In such cases the parse tree is not modified and no instrumentation will be emitted from that node. `ENode()` inserts a new enode under the following conditions:

- The configuration file specifically calls for the event
- The configuration file calls for an event selection that requires run time checking (described later)
- The configuration file calls for an event selection that requires syntactic analysis or semantic analysis that must be postponed until the entire tree is constructed.

2.3 Type information and Symbol Tables

In addition to parse tree construction, CCI maintains symbol table information for type propagation and symbol storage, similar to any compiler. Type propagation is important for event generation because 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 event codes. 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 Instrumentation Output

Once CCI has built the parse tree and determined what events can be instrumented, it traverses the parse tree emitting event generation code along with source code. An example of this is shown in Figure 3; in the figure, all events in the basis set are instrumented.

In addition to calls to the event macro `EV()`, there are a few other items which deserve attention. First, CCI creates temporary variables to hold intermediate results of instrumented expressions. This is necessary to ensure expressions are evaluated only once, as in the original source code, ensuring no side effects and maintaining overall semantic behavior. A more detailed discussion of temporary variables is provided below. Next,

```
Before Instrumentation:
a = b + c;

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

Figure 3: Example of CCI's Instrumentation

as shown in the preceding instrumentation example, even for a simple expression such as `a = b + c`, there is a problem of code explosion. CCI's configurability addresses this issue by producing 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);
```

It is easier to see in this instrumentation example how CCI inserts code into individual expressions without changing their result. CCI's instrumentation technique takes advantage of the C comma operator. First `b+c` is evaluated and stored into a temporary variable, `_Tcci_0`. `_Tcci_0` is passed as the value of the event macro and subsequently provided as the result of the expression, which is assigned to the variable `a`.

2.5 Temporary Variables

In Figure 3 many temporary variables, those variables whose names begin with the prefix `_Tcci_`, were created. As discussed earlier, temporary variables are created in order to hold intermediate values of expressions that must be passed as events. These temporary variables are allocated as automatic storage in the function they are executing but are ensured to be allocated first on the stack. The purpose of this is to reduce the chances of temporary variable memory overwrites from an errant target program.

In addition, temporary variables are useful for conveying type information to monitors that use CCI. For example, the Alamo framework utilizes the debugging symbol information for the instrumented program. A cast expression has no type listed in the stabs section of the object file, but a temporary variable does. Thus for cast expressions, temporary variables expose semantic information that is otherwise hidden in object level instrumentation systems. An example of this mechanism is a call to `malloc`. Values returned by `malloc` are typecast to a pointer to some type of object. This information is available to the Alamo framework because of the instrumentation provided by CCI. Figure 4

shows an example of how the typecast instrumentation works.

Before Instrumentation

```
node *n = (node *)malloc(sizeof(node));
```

After Instrumentation

```
node *_Tcci_0;
node *n = (EV(E_Cast,(_Tcci_0 =
   (node *)malloc(sizeof(node))),&_Tcci_0),_Tcci_0);
```

Figure 4: Cast Expression Example

For the `E_Cast` event[1], CCI generates a temporary variable of the type to which the value is cast, and assigns that variable to the rest of the expression. The address of the temporary is the `E_Cast` event's value. The monitor, if it has access to the target program debugging symbol information, can determine the type of the cast expression. Finally, `_Tcci_0` is the result of the entire expression which is assigned to `n`.

Another example where temporary variables are quite useful is for the `E_Block` event. This event marks the beginning of the any block or procedure definition. For example, given a function `foo` the `E_Block` event would look like the following:

```
void foo(int y) {
   int _Tcci_Start = (EV(E_Block,0),0);
   int x;

   y = boo();
   X +=y;
}
```

The `E_Block` event is generated before any parameters or local variables are modified. In addition, it marks the beginning of a function call.

3 Configuration

Without configuration, CCI produces huge amounts of code intrusion along with a significant performance penalty, discussed in section 4. Moreover, 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 that is supplied for standard headers and libraries, raises the semantic level of events for all applications that use those modules.

To reduce the volume of events and create high level events, CCI uses *static* filtering, inserting only selected events at compile time. Individual monitors and monitor frameworks such as the Alamo system may provide further dynamic filtering, testing for particular conditions at run time.

CCI's configuration language has facilities for selecting, refining, and composing basis events into higher level events. In order to perform these operations, the monitor writer must know and understand the primitive events that are defined in the basis set, described in section 2.1. This required knowledge is 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.

In addition, 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 refers to event selections that have are local to a particular function or range of line numbers, and apply instrumentation only within their scope. 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 files were really contained in one large file.

3.1 Event Selection

As described in section 2.1, CCI has a basis set of approximately 160 event codes that represent the lowest level events available to describe program behavior. The basis set contains event codes for program behaviors such as assignment operations, variable dereference, procedure calls, control flow, array index, and structure accesses. During configuration, monitor writers select desired events from CCI's basis set. To make event selection easier, CCI has many built in sets of event codes. The sets are used purely as a notational convenience. For example, `E_Add` is the set of all addition events regardless of type.

[1]This example shows only cast events. Other basis events that CCI can produce for this expression include function reference, parameters, call, return, and assignment of the result.

3.2 Event set selection and definitions

An event set selection is a comma separated list of event names terminated by a semicolon. Also, an event set can consist of a single event. The grammar[2] for this is specified as

```
EventSet  => EventList ;
EventList => Event | EventList , Event
```

In the above grammar `Event` is any one of the basis events or a *set definition*. A set definition is simply a name used to denote a set. An example of a set definition follows:

```
E_Math = (E_Add,E_Sub,E_Div,E_Mult,E_Mod);
```

This definition assigns the identifier `E_Math` to the set of all mathematical operations involving operators. However, set definitions do *not* tell CCI to generate instrumentation for those events. To do this, the user must select the event as follows:
`E_Math;`
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, if the function `heap_sort` is specified it would look as follows:

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

If the function is declared as static, there may be many functions with the same name but in different source files. In CCI, the user delineates which function to instrument by using the source file's name as in the following example:

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

In this example, only the function `heap_sort` in the file `foo.c` will get instrumented.

As mentioned earlier, CCI allows the monitor or framework that uses CCI to define the invocation mechanism. This leaves many of the performance considerations that are associated with instrumentation up to the monitor writer. However, CCI aids performance by using static filtering and building more high level events through special filtering techniques. These techniques, called *event masks*, are described below.

[2] This is a simplified grammar. The actual grammar contains more features but would obscure the example.

3.3 Event Masks

In CCI, there are two kinds of masks, *type masks* and *value masks* which are both applied to the event itself. As discussed earlier, these mask types perform a variety of filtering services. Some of the filtering is done at compile time and some must be done dynamically. Below is a discussion of these filters, their use, and the nature of the filtering, static or dynamic.

3.3.1 Type Masks

A type mask, a static filter, filters an event based on the event value's type which is determined at compile time. For example, if a monitor writer wants to instrument only assignments to structures that are of type `struct foo`, the user masks the assignment to structure event, `E_Assigns`, as follows:

```
E_Assigns{struct foo};
```

In the above example, we showed how type masking can be applied to particular basis event. However, combining event set definitions with type masking has useful implications. Recall that the `E_Math` event name describes the set of all mathematical operator events, `E_Add`, `E_Sub`, ..., etc. Each member of the `E_Math` set is also an event name that defines yet another set. For example, the `E_Add` set consists of the following basis events, `E_Addi`, `E_Addf`, `E_Addc`, ..., etc. Now a monitor writer may want to receive only events from the `E_Math` set that are of floating point type. This is simply given by `E_Math{float};`. Now if a floating point operations occurs, CCI will generate the basis event that is a member of the `E_Math` set.

An important feature of masks is that they may *refine* the event. 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. In addition, this same refining technique can be used to map many events into one event. For example,
`E_Math{float = E_FloatMath}`.

Here, for every event generated that is a basis event member of the `E_Math` set whose type is masked by float will cause a `E_FloatMath` event to be generated. Thus instead of receiving an `E_Addf` or `E_Subf` event, the monitor receives only one event: `E_Math`.

In addition to simple events as those described above, type masking can also be applied to functions. For example, if the monitor writer wants to receive only `E_Pcall` events for functions that take `struct foo *` as their first parameter and return `struct foo *`, this is given by

```
E_Pcall((struct foo *(struct foo *,...))=E_FooWatch)
```

The value of type masks in configuration is considerable. Type masks enable CCI to generate high level events. They allow for the reduction of events in a way that is not available to machine level instrumention. Type information is available during parsing and type filtering is performed at compile time, reducing run time costs.

3.3.2 Value Masks

Perhaps more powerful than type masks are value masks which filter an event based on its value. CCI performs value masking using static information available at compile time. Monitors or monitor frameworks may augment CCI's information by performing further value masking at runtime.

As an example of CCI's facilities, suppose that a monitor writer wants to instrument only calls to `malloc`. If no filtering is available, the monitor would have to receive all `E_Pcall` events. The address of the function that is going to be called, `E_Pcall`'s event value, must be checked by the monitor for each function call. However, by using a value mask filter applied to an `E_Pcall` event, CCI filters all `E_Pcall`s whose value does not equal that of the value mask. For example, to only report calls to `malloc`, the event selection is `E_Pcall{malloc}`.

Often, however, the user is not just interested in masking an event for only one value but many values. In CCI this is simple. Suppose the user wants to monitor calls to all library memory allocation functions. To do this the user types

```
E_Pcall{malloc,calloc,realloc};
```

The above example shows how to construct high level events from simple value masks. In this example, `E_Pcall` events are still generated, albeit only for the functions `malloc`, `calloc`, and `realloc`. The problem with this is that the monitor still doesn't know which function was called; it could be any of the three functions. CCI solves this problem by allowing the monitor writer to refine the basis event by substituting one or more new event codes for the filtered basis event. To create events that are unique to the value masks given in the above example the user types:

```
E_Pcall{malloc=E_Malloc,calloc=E_Calloc,
        realloc=E_Realloc};
```

Now instead of an `E_Pcall` event being generated when `malloc`, `calloc`, and `realloc` are called, the new events `E_Malloc`, `E_Calloc`, and `E_Realloc` are generated respectively.

Another example of value masks is that of variable modification trapping. To set up a trap on assignments to variables `node`, `child` and `parent`, the user types

```
E_Assign{node,child,parent};
```

Variable modification trapping is costly in many monitoring frameworks, because checking all accesses to variables (and pointers to variables) is prohibitive. Trapping variable modifications for multiple variables exacerbates the performance cost.

In the presence of aliasing, variable modification trapping can only be solved at run time in general. Static analysis can reduce the performance cost of this dynamic analysis. CCI's static analysis is presently unsophisticated. The general solution to this problem requires an action from the monitor at run time, such as checking pointer dereferences against a hash table of memory addresses of interest.

4 Performance of Instrumented Code

Comprehensive instrumentation such as is provided by CCI is very intrusive, both in terms of code size explosion and execution speed. The purpose of CCI's configuration facility is to reduce the costs of automatic instrumentation to acceptable levels. While acceptable for the intended purpose of supporting interactive dynamic analysis and program steering tools such as program visualizers, CCI's instrumentation may not be suitable for very large programs or monitoring programs with real time performance constraints.

Performance was measured using three sample applications listed below. This paper gives overall results for the three programs, and detailed results for the first application; additional details are available in [7].

`concord.c` – builds and writes out a sorted concordance of words that occur in a text file; ran on a 90KB input file.

`life.c` – Conway's game of life, 20x20 board, one spinner; ran for 30 iterations.

`laplace.c` – solution to Laplace's equation at a point in an annulus via a Monte Carlo method; ran with 1000 iterations.

These sample applications were instrumented with configurations that comprised a range of selected events:

- all basis events,
- categories of events such as procedure calls or structure accesses, and
- specific events based on value masks, such as instrumentation of the `malloc(3)` library.

Each range of events were executed with a range of possible monitor types characterized by their macro definitions:

- a minimal monitor consisting of in-line counters at each event,
- a typical monitor that calls a procedure to execute monitor code at each event, and
- a coroutine monitor that performs dynamic filtering of events and a lightweight thread switch at each event.

The monitors in all cases presented below did nothing but count the number of events of each type; the goal was to measure the cost of instrumentation introduced by CCI. The costs of the monitor performing dynamic analysis or rendering a visualization are not considered.

4.1 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 for the sample programs ranged from 3-5 when tiny macros and narrow event selections were 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, when a large macro was used on configurations of varying sizes, the measured code explosions ranged from 10 to 60! The larger the macro definition cost M, the more critical it becomes to provide higher level configurations 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.

4.2 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 ranged from 2-4 for in-line counters, and from 4-9 for monitors that execute as a set of callback routines.

The slowdown introduced by the coroutines in the Alamo C framework shows the most extreme case for CCI's configuration facilities. Figure 5 shows the performance numbers obtained by executing `concord.c` on various configurations using the Alamo framework. M is gigantic, reflecting the runtime cost of a lightweight context switch as compared with the small cost of typical C operators and control structures. By configuring events to select specific types and operations corresponding to abstractions such as those provided by sets of library functions, the execution slowdown imposed drops from three orders of magnitude down to under one order of magnitude. As instrumentation becomes more selective, the configuration cost C becomes quite small.

Configuration	Time(sec)	Description
none	0.31	normal program run
vars	849.9	memory references
math	241.5	arithmetic operators
fcalls	153.6	all library calls
string.h	53.6	string.h calls
malloc.h	1.6	malloc.h calls

Figure 5: The effects of configuration on instrumentation runtime cost.

5 Related Work

Automatic software instrumentation systems can generally be classified along several dimensions, including: compile time versus run time, machine level versus high

level, and standalone versus integrated into a specific compiler or interpreter. CCI can be characterized as compile time, high level, and standalone. This section considers a number of related systems that illustrate a variety of techniques for software instrumentation.

Source level debuggers exemplify the most successful category of monitoring systems, such as [10], [8], [9] . Such systems instrument programs at run time for specific locations identified by the user. They are especially suited to tasks in which the nature or location of the bug is understood. CCI, in the context of the Alamo architecture, helps simplify the construction of monitor programs to automate the task of locating problems that are *not* understood. Constructing such monitors on top of source level debuggers has been done successfully in the DALEK[12] system, but conventional debuggers use a two-process execution model with heavyweight context switches that has even worse performance problems than those described above for the Alamo C framework.

Monitoring systems such as EEL[4] and ATOM[6] instrument code at the object level. Object code modification has applications including emulation, observation, and optimization of hardware architectures [4]. If a monitor needs instruction level instrumentation, EEL and ATOM are suitable tools. These systems can instrument higher level behavior to some degree, such as function calls and accesses on particular data types.

The ATOM system offers a programmable approach to instrumentation. It requires that the user specify where to instrument the program and how to instrument it. Instrumentation in ATOM is done through an user written instrumentation function that makes calls to program analysis functions. These functions analyze and locate various language independent constructs such as function calls and basic blocks, throughout the target program. Using these analysis functions, the instrumentation function sets up which function to call when a particular instrumented event occurs.

Another interesting monitoring system is Dynascope[3]. Dynascope does not instrument the source program. Instead Dynascope uses a model where the monitor interrupts the target when it wants some kind of information. Dynascope uses two main components a *director* and *executor*. The director consists of a user program, along with the Dynascope client library, that together serve as a monitor. The monitor sends *directing primitives* to the directing server which is in the same address space as the program being monitored. Together they are called the executor. The director and executor communicate via monitoring and control primitives that operate across networks using TCP/IP. The executor receives a primitive from the director which causes control to be transfered to the directing server. There, the directing server can give access to program state and variables and control the execution of the target program. Dynascope also works at the object code level in that all event information relates to assembly language level information[3]. Dynascope is used for execution steering, debugging and relative debugging.

Another system, called PMMS[5], automatically instruments a very high level language AP5. In PMMS, the user supplies a configuration in the form of relational calculus queries to describe the information about the program's performance that is of interest. Then PMMS determines where to insert instrumentation based on those queries. However, the selection language is directed toward queries regarding peformance details and not towards visualization and debugging. CCI, like PMMS, uses a configuration file to filter events out before instrumentation, but CCI's configuration language guides the insertion of instrumentation based on syntactic and semantic specifications.

CCI's emphasis on type information and the extraction of abstract data type behavior from C structures and library functions was influenced by UWPI[11], a compiler for a simplified Pascal grammar that analyzes the source program and automatically builds a dynamically updating view of the program's execution. UWPI uses syntactic and simple dataflow analysis to determine which structure is the most "complex" and uses that structure as the backdrop for the rest of the program's illustration. Given the right configuration, CCI extracts comparable information from programs written in a standard compiled systems language, and delivers events to arbitrary monitors by means of a user defined macro interface.

6 Conclusion

Monitors are difficult to write because the monitor writer usually must have a detailed understanding of the system to be monitored. When the information received from instrumentation is very low level, the monitor writer must spend time and effort extracting high level information. Moreover, monitors always incur a performance cost, which renders many execution models unsuitable, especially those which minimize intrusion.

CCI was developed to address several of these problems. CCI addresses the difficulty of writing monitors by providing an instrumentation tool that can help abstract, although not completely, the low level nature of instrumentation. CCI produces instrumentation that is both flexible and high level and has facilities for improving performance. Using CCI to instrument programs, monitor writing is simpler and more accessible. As a result, visualization tools and debugging mechanisms can

be explored more easily.

CCI reduces code explosion by providing configuration. Through configuration a monitor writer can select which 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.

Along with CCI's set definitions and selections, its type and value masks allow for powerful behavioral expressions. The declarative configuration language prevents the monitor writer from having to learn a new programming language or set of library function calls. In addition, configurations for common C libraries can be shared, allowing sophisticated instrumentations of higher level program behavior that are equally applicable to any C program that uses the libraries in question.

References

[1] Weiming Gu, Jeffery Vetter and Karsten Schwan "An Annotated Bibliography of Interactive Program Steering" *ACM Sigplan Notices*, Volume 29, No 9, pp. 140-148, September 1994.

[2] Clinton Jeffery, Wenyi Zhou, Kevin S. Templer, and Michael Brazell "A Lightweight Architecture for Program Execution Monitoring" *Proceedings of the ACM Workshop on Program Analysis for Software Tools and Engineering, PASTE'98*, Montreal, June 1998, to appear in SIGPLAN Notices.

[3] Rok Sosič "The Dynascope Directing Server: Design and Implementation" *Computing Systems*, Vol 8., No. 2, pp. 108-134, Spring 1995.

[4] James R. Larus and Eric Schnarr, "EEL: Machine-Independent Executable Editing," *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, SIGPLAN Notices, Vol 30., No. 6, pp. 291-300, June 1995.

[5] Yingsha Liao and Donald Cohen, "A Specificational Approach to High Level Program Monitoring and Measuring," *IEEE Trans. on Software Engineering*, Vol 18, No. 11, Nov. pp. 969-978, Nov. 1992.

[6] Amitabh Srivastava "ATOM: A System for Building Customized Program Analysis Tools" *WRL Research Report 94/2* Western Research Laboratory, 250 University Avenue Palo Alto, California 94301, March 1994.

[7] Kevin S. Templer, "Implementation of a Configurable C Instrumentation Tool" *Master's Thesis*, Division of Computer Science, University of Texas at San Antonio, May 1998, `www.cs.utsa.edu/research/alamo/cci.ps.gz`.

[8] M. Golan and D. R. Hanson, DUEL — A Very High-Level Debugging Language, *Proceedings of the Winter USENIX Technical Conference*, 107-117, San Diego, CA, Jan. 1993.

[9] David R. Hanson and Jeffrey L. Korn, A Simple and Extensible Graphical Debugger, *Proceedings of the 1997 USENIX Annual Technical Conference*, 173-184, Anaheim, CA, Jan. 1997.

[10] M. A. Linton, The Evolution of Dbx, *USENIX Summer Conference*, 211-220, June 11-15. 1990.

[11] R. R. Henry, K. Whaley and B. Forstall, The University of Washington Illustrating Compiler, *Proc. ACM SIGPLAN'90*, 223-233, June 1990.

[12] R. A. Olsson, R. H. Crawford, and W. W. Ho, Dalek: A GNU, Improved Programmable Debugger, *USENIX Summer '90 Conference*, 221-231, USENIX Association, June 1990.