IMPLEMENTATION OF
A CONFIGURABLE C INSTRUMENTATION TOOL

by

KEVIN SEBASTIAN TEMPLER, B.S.

THESIS

Presented to the Graduate Faculty of
The University of Texas at San Antonio
in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
May 1998

Acknowledgements

I wish to thank the members of my committee, Dr. Clinton Jeffery, Dr. Kay Robbins, and Dr. Samir Das for their input and support. In particular, I am very grateful to Dr. Jeffery, my advisor and friend, who insisted on only the very best from me while "putting up" with my crazy schedule.

I would also like to thank Wenyi Zhou, who has already graduated and is working. During the first years of my research she was, and still is, a good friend and the greatest and most fun colleague I've ever known. In addition, my thanks goes to my boss Dr. Schroeder who has been a good friend and supporter of my graduate studies even when that means I am not working on REACH Interface Author.

My parents deserve many thanks. Their support is beyond mearesurement. Also, I want to specifically thank my Dad who started me out on computers when I was 12 and said, "If you learn this, you'll do great things." In addition, I would like to thank my brother, Scott, for always being caught-up on the latest hardware and operating systems so I didn't have to.

Finally, I wish to thank my wife whose patience and understanding is unending. Thank you Patricia for always being there and supporting me and listening to various ramblings about my work.

KEVIN SEBASTIAN TEMPLER

The University of Texas at San Antonio
April 30, 1998

Abstract

One of the most difficult aspects of dynamic analysis is extrapolating behavioral information from the program to be debugged or visualized. This thesis describes the design and implementation of a Configurable C Instrumentation tool, called CCI, which addresses this difficulty. CCI inserts event generation code into a program while preserving the original run-time behavior, other than reducing its speed. The event generation mechanism is user-defined by the framework which uses CCI. CCI addresses the fact that inserting code causes code explosion and reduced execution speeds by providing configuration facilities. Through a simple event set grammar, the user can specify instrumentation for those events they are interested in. Moreover, the user can create new high level events from pre-existing low-level events. Configurability is the major research contribution of CCI: it reduces code explosion and the total number of generated events while increasing the speed of the instrumented code. Its configuration facilities make CCI a viable tool for automatic instrumentation in an execution monitoring framework.

Contents

Chapter 1

Introduction

Understanding how a program behaves is a critical part of debugging. In addition, as the complexity of the program increases, it becomes increasingly important for the programmer to able to understand its behavior. An execution monitor is a program that monitors or observes another program. Examples of execution monitors are debuggers, profilers and visualization systems. While each of these kinds of execution monitors perform different tasks they all attempt to provide information about a program's behavior – all perform some kind of debugging.

Although research in improving debuggers has been steady, the *process* of debugging has not changed significantly. The process of debugging entails using print statments, source-level debuggers and/or profilers. Currently, source-level debuggers provide control of execution and the ability to inspect program state. Some debuggers such as DUEL [Gola93] or DALEK [Olss90] have high-level languages that provide for more sophisticated methods of program state inspection and applying breakpoints.

Debugging often requires high-level views of program state. For example, debugging operations on tree structures typically involves analyzing an entire collection of objects which is tedious when using a source level debugger – the programmer often

finds it difficult to maintain a mental model of many structures and their relationships. In addition, information about a program's resource usage, dynamic data flow and control flow and its interactions with internal structures is difficult to obtain using source-level debuggers. These high-level debugging requirements are simply not within the domain of interactive debuggers. Thus, programmers need higher level debugging techniques to address the increasing complexity of program behavior.

If programmers can observe information about a program's behavior graphically and dynamically, they can better ascertain its overall behavior and functionality, or lack thereof. High-level debugging involves using an execution monitor framework that provides a tool-chest of execution monitors that help observe a program's behavior, perhaps providing a gestalt view of a program. As the programmer gains more understanding using high-level tools, they can use other more specific tools, employing more specific monitors, such as an interactive debugger. Throughout this thesis, the term execution monitor is used to refer to monitors that are capable of high-level visualization, rahter than a traditional source-level debugger.

Unfortunately, execution monitor research is not making progress at comparable speeds to increases in application complexity. The reasons for this are two-fold. First, there are few frameworks which provide for execution monitor research. Of those that exist there are none that provide for high-level execution monitors as described above [1], especially for C. Second, writing monitors is generally difficult. Writing a monitor requires detailed knowledge of the operating system and other machine level details. In addition, the monitor writer must spend considerable time designing and implementing the best method of presenting information from the execution stream. This alone is formidable. Consequently, monitor writing and development is obviated by more traditional debugging methodologies such as using source-level debuggers, profiling, or hand

[1] The Icon Monitor Framework [Jeff94] is an exception, providing facilities for exploratory development of high-level execution monitors on programs written in Icon.

insertion of `printfs`.

1.1 Motivation

The C programming language is a difficult programming language. It is difficult because it is low-level and, as a result, is more error prone than higher level languages. Due to its difficulty of use and its tendency for programming errors, it needs a more robust debugging environment. With an effective execution monitoring framework, it is possible that more interesting and powerful execution monitors can be developed that will help the C programmer visualize and debug their programs more efficiently and easily. In addition, building an effective execution monitor framework for the ubiquitous language C, allows for exploratory research of visualization tools.

An environment where writing execution monitors is fast and simple requires the following:

1. An instrumentation system
2. A monitoring system
3. A method for displaying the information obtained from the monitor.

Figure 1.1 shows the various components of an execution monitor. A description of each component follows:

The Target Program

The target program is the program being monitored, its source code may or may not be available [2]. The instrumentation system collects information about the target program's behavior.

[2] If using CCI, the source must be available.

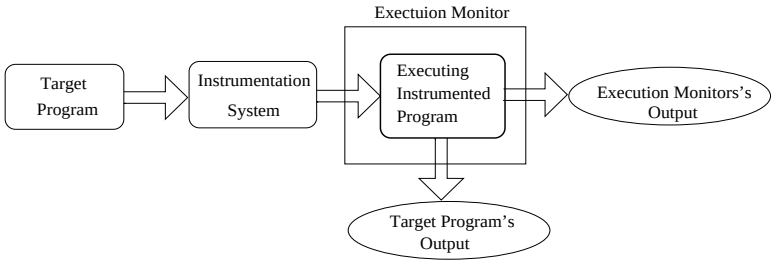


Figure 1.1: An execution monitoring environment.

The Instrumentation System

Developing an instrumentation system one must consider the language level for the instrumentation. The language level is important because it relates to the amount of behavioral information available for instrumentation. In machine or assembly language, for example, each instruction contains miniscule amounts of semantic behavior, while C, a higher level language, provides more semantic information per line of source code. Generally, the higher the language level, the more semantic information available per line of source code. Consequently, to extrapolate the maximum amount of behavior from a program, instrumentation should be tightly coupled to the source language.

The Monitoring System

The monitoring system collects the information produced by the instrumentation system. This system typically runs concurrently with or as a part of the target program. This is depicted in Figure 1.1 by the execution monitor box. In a debugger, for example, the monitoring and instrumentation system are tightly integrated.

Information Display

The method for displaying the information can be in the form of a graphical or textual display which is either static or dynamic. Or the information can be recorded in a log file for later analysis.

This thesis presents a Configurable C Instrumentation tool, named CCI, that automatically instruments C programs. In addition to automatic instrumentation, CCI provides a method for configuration that reduces the penalties normally associated with source code instrumentation and provides high-level behavioral information to a monitor. CCI is portable and is designed to be used with a monitoring framework or by itself.

The rest of this chapter first provides some useful definitions, followed by an overview which briefly describes the method of instrumentation used by CCI and how it is used in a larger framework called Alamo. In addition, related work is explored. The chapters that follow discuss the design of CCI, its implementation, and an analysis of its performance. The final chapter draws conclusions and discusses limitations and future work.

1.2 Definitions

intrusion – When a program is instrumented, it must be perturbed in some way by adding additional code of some kind to "trap" and report a particular behavior in the target program. The goal of any instrumentation system is to reduce intrusion as much as possible.

user– A user is one who is using the instrumentation tool and/or an execution monitor. Often the user is the person debugging the target program.

1.3 An Overview of Instrumentation

The goal of instrumentation is to provide information from a particular program execution to a monitor. CCI describes execution information in terms of events. An event is any atomic unit of program behavior. As described earlier the definition of a "unit of

program behavior" changes depending on the language of the program that is being instrumented. For example, each assembly language instruction could be considered a unit of behavior; whereas, in C, a structure assignment or function call could also be considered a unit of behavior. One of the major challenges of CCI is to derive high-level behavioral information from a relatively low-level language, C.

1.3.1 Instrumentation Methods

There are various methods of instrumenting a program. The program can be compiled into machine language, then loaded into memory where instrumentation is inserted at the machine code level. Some execution monitoring systems use this approach [Holl90], [Olss90]. An advantage of instrumenting at the machine code level is that it is language independent because the instrumentation does not consider the source language.

Another method is to insert instrumentation into the source code. This method has two advantages. First, it is more portable. Second, more semantic information can be extrapolated from the source than is possible from assembly level instrumentation.

Some monitors obtain all their information in the form of log files and then perform post-mortem analysis (e.g. gprof, etc.). Other monitoring systems dynamically capture information at run-time. Dynamic run-time monitoring systems must be executing, in some manner, concurrently with the instrumented program using some method for communicating the information from instrumentation. Some monitoring systems use operating system debugging traps to handle the event stream. Others may use simple function calls in the same address space. Still other monitors execute using context switches. All of these methods have various tradeoffs and will be discussed later.

Once the monitor obtains the information from the instrumentation system, it must have some method of presenting this information to the user. Again, there are various

methods for presenting information. One trivial method is to store the information gathered by the monitor into a trace file. This, however, is mostly unacceptable because of the enormous amount of information that is typically produced from execution monitoring. Moreover, even if the data were manageable, there are analyses that can not easily be done post-mortem because there is no opportunity to access and manipulate program state. While post-mortem analysis tools can reveal various kinds of program behavior, they are not meant to address those debugging needs where access to program state is essential. Another method of presenting information is to display the information dynamically either graphically or textually. Graphically displaying information is preferred because of the ability to display more and clearer information than that available using textual techniques.

1.3.2 Alamo and CCI

Alamo [Jeff96], A Lightweight Architecture for MOnitoring, is a framework that provides for exploratory development of execution monitors for the C programming language. This framework encourages execution monitor development by abstracting the details, program state access and event acquisition, typically associated with monitor-writing from the developer.

The organization of Alamo is shown in Figure 1.2. The object labeled "1" represents CCI. The instrumentation inserted by CCI consists of event generation code that uses a simple event model where each generated event contains an event code and event value. A simple example of an event might be a call to a function or an array access. CCI does not define the event generation code but leaves its implementation up to the framework which uses it. In Alamo, event generation is implemented using light-weight context switches. The context switching mechanism allows the target program and the execution monitor to pass information back and forth using a coroutine model.

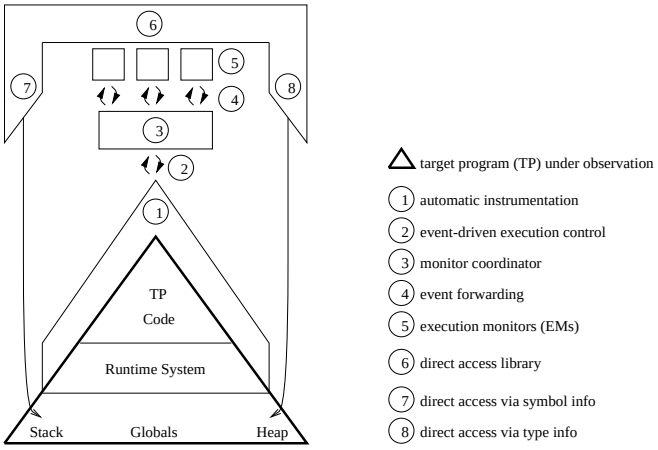


Figure 1.2: The organization of the Alamo Framework

The component responsible for coordinating the communication between monitor and target program is called the Alamo Monitor Executive [Zhou96]. This module is shown in Figure 1.2 by the numbers two through eight. AME is responsible for loading the instrumented and compiled target program and any execution monitors into the same address space. The program then starts the execution of the monitors and the target program. All execution monitors begin by setting up an *event mask* which specifies a bit vector containing all events the monitor desires. Then each monitor enters into a loop which waits for an event. This is not idle waiting but instead causes a context switch to the target program. The target program begins execution and, when an event is encountered, a context-switch is performed, returning control back to the AME. The AME then forwards the event to each of the monitors *only* if their event mask is set to receive that type of event.

Once control has been returned to the monitor the monitor processes the event and can access or modify program state. In addition, the monitor writer can depict the current state of the program, based on the event stream received so far, graphically and dynamically.

One example application of Alamo is a tree visualizer [Braz97]. This program provides a monitor that helps to visualize any tree structure that is created in the target program. It may be useful, for teaching purposes, to simply write a program that builds a tree and let the monitor display it graphically. Or for the purposes of debugging, the programmer can verify that the nodes of the tree are correctly being connected. Moreover, if a programmer must write a tree balancing program, it may be difficult to verify that it "really" works without constructing a tree walking program that outputs each node. In addition, the programmer may already know that the tree is balanced but be interested in how often the tree must re-balance itself. These particular examples are only the beginning of potential uses for program visualization. Example output of the tree visualizer is presented in Figure 1.3.

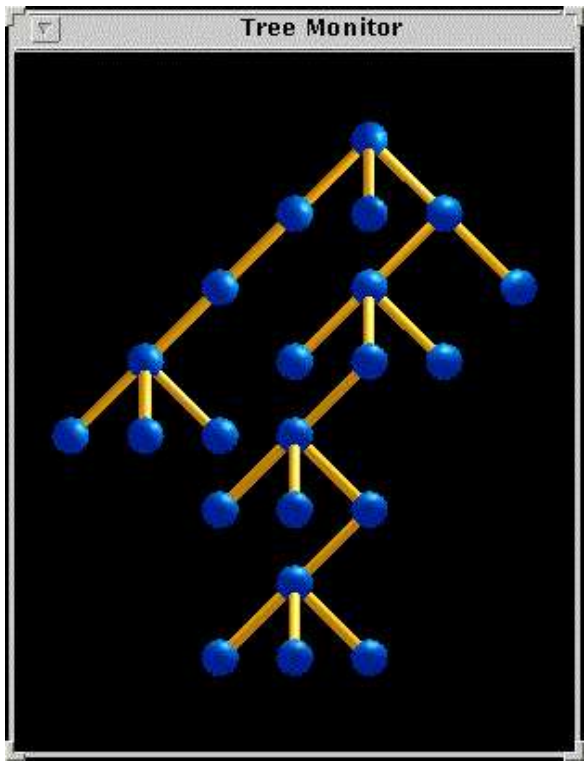


Figure 1.3: Tree Visualization Execution Monitor.

This program attempts to detect which structures are tree structures and, at run-time, graphically display the tree and its changes. This particular program does not use CCI's configurability [3]; instead it uses the lowest-level event stream produced by CCI.

The purpose for creating Alamo is to make monitor writing easier so that visualization and execution monitoring research becomes more viable. It is hoped that because monitor writing is more simple it will encourage the development of more interesting visualization tools. As a result programmers that need these tools will not have to write their own but can use a suite of already-written monitors, each developed to visualize specific kinds of behavior of a program.

1.4 Related Work

An important consideration of instrumentation is when it is performed. Instrumentation is inserted either before or during compile time or just before or just before or during run-time. The chosen method determines what kinds of information are easily obtained from the target program. Moreover, the method also dictates what level of debugging the monitoring system is capable of.

Instrumentation is added either automatically, semi-automatically or manually. Automatic instrumentation is done by a program which inserts instrumentation without user intervention [4]. Semi-automatic instrumentation is when a small portion of code, written typically by the user, directs where the instrumentation is inserted. Manual instrumentation is where the user inserts instrumentation code by hand into the target program, for example using `printf` or where the code is annotated by macros that

[3] CCI's configuration component was not yet implemented at the time the tree visualizer was developed.

[4] In some systems it is still necessary for the user to specify what should be instrumented, thus it is not entirely automatic.

perform instrumentation.

The information gathered from instrumentation is also contributes to the kind and level of monitoring that is possible. Many instrumentation systems are quite low-level, reporting machine-level information. On the other extreme, an instrumentation system can report high-level information such as the access or manipulation of a link list [Jeff93]. The goal of CCI is to be able to extract from the C programming language, information similar to the high-level information available from monitoring languages like Icon or other high-level interpretive languages.

1.4.1 Run-time Instrumentation

DBX [Lint90], a typical source level debugger, instruments a target program by using operating system traps at run-time to stop at some user-defined breakpoint. DBX uses an event model based on low-level information such as the particular value of the program counter or the particular function's address. DBX provides very strong intrusion control by running as a separate process and allowing the program to run at (potentially) full speed. DBX provides for some programmable instrumentation as well. For example, the user can write simple programming constructs to specify when DBX should stop executing the target program and return control to the debugger. However, as mentioned earlier, DBX and source level debuggers still require the user to insert break points and to have an "idea" of where the bug is they are trying to fix. Moreover, the views provided by a source-level debugger are still fine-grained enough to not change the fundamental method of debugging.

Dalek [Olss90] and DUEL [Gola93] are two noteworthy debugging environments that extend traditional source-level debugging techniques. Dalek, for example, runs on top of gdb and is a programmable debugger. Dalek provides a C-like expression language for evaluating program state, and it also provides for an event trapping mechanism. Dalek

uses a machine level event mechanism for instrumentation. DUEL also runs on top of gdb and provides a high-level debugging language that uses features found in Icon such as goal-directed evaluation and generators. These features provide for economy of expression while being able to iterate through structures within the target program. While each of these debuggers provide the programmer with powerful facilities for debugging, they are not capable of significantly helping the programmer find bugs when the location and nature of the bug are unknown. Higher level tools are needed to address this problem.

[Holl90] describes a method of instrumenting programs during execution of parallel programs. In order to reduce the amount of instrumentation, programs are instrumented at run-time. This process involves replacing one or more machine code instructions with a call to a base instrumentation routine, created for each instance of instrumentation. The base routine performs the necessary housekeeping of saving registers and program state, then calls another smaller routine which actually performs any event monitoring code. Instrumenting at run-time has an advantage in that it can reduce the number of generated events and intrusion, however, in this case the level of instrumentation is at the machine-code level. Consequently, this reduces the domain of applicable execution monitors because at the machine code level there is little semantic information available as compared with its source code counterpart.

1.4.2 Machine Level Instrumentation

ATOM [Sriv94] is a framework for building customized analysis tools on object-modules. ATOM is built using OM and is language and compiler independent. In ATOM, the user supplies an instrumentation program specifying what to instrument. All routines necessary for instrumentation and program state inspection are located in the address space of the target program. Instrumentation routines are supported by an instrumentation library that provides machine-independent methods of accessing various components of

the target program. However, all instrumentation is at the machine language level. Thus the level of abstraction is limited similar to source level debuggers. In addition, similar to run-time debuggers, the user must specify where and how to instrument the program.

1.4.3 Instrumenting Compilers

The University of Washington Illustrating Compiler (UWPI) operates on a simplified Pascal grammar [Henr90]. This system analyzes the program and automatically builds a dynamically updating view of the program's execution. UWPI uses syntactic and simple data flow analysis to determine which structure is the most "complex" and uses that structure as the backdrop for the rest of the program's illustration. UWPI uses its analysis to apply predefined template views onto the target program. Once the compiler analyzes all information, it inserts procedure calls that call and communicate current state and run-time information to the layout component of UWPI.

1.4.4 Interpreter Instrumentation

Dynascope [Sosi95] is an execution monitor that has two main components: the director and the directing server. The director consists of a user program and client library that communicates with the directing server. These two components communicate using events. Events are classified as any location modified by the user program. Location refers to registers or memory addresses, so the event stream is extremely fine and low-level. The information from the events is formed by executing the machine instructions in a virtual machine environment or simulating the instruction directly, if possible, and then reflects the results to the director. The director communicates with the directing server using interprocess communication. The director, a separate process, attaches itself to the target program and directing server which are located in the same address space. This feature allows many different directors (or monitors) to connect with the target program making

it quite useful for applying different monitors at run-time. While this system is quite powerful because it is capable of working in heterogeneous distributed environments, its event information is low-level precluding many high-level views of program behavior.

Icon is a language that provides significant monitoring capabilities. Icon's virtual machine contains built-in instrumentation at key points in the byte code interpreter providing for high-level monitoring of a program. The monitor and the target program run concurrently and information is transferred in the form of events using light-weight context switches. Quite a few execution monitoring tools have been developed and have been used successfully to find bugs that would have been difficult to find otherwise [Jeff93].

1.4.5 Goals of this Research

The related work points to the fact that there is a need for more advanced methods of debugging and visualization. It is also important to reduce intrusion and minimize the run-time performance effects of instrumentation. These observations motivate CCI's goals:

1. Instrument C programs to provide a "basic" level of behavioral information in C programs.

2. To reduce intrusion, provide configuration methods for the control and selection of instrumentation.

3. Provide additional configuration facilities that will raise the level of information obtained from C programs, possibly approaching that of high-level languages.

Chapter 2

Design of CCI

This chapter discusses the overall design of CCI, describing each of its major components. The first section covers CCI's functional organization. The second section covers the "Basis Set," a notion of the granularity of observable behavior. The final section covers configuration, arguably the most important design component of CCI.

2.1 Organization of CCI

CCI instruments a target program by inserting C code at key points throughout the program. The instrumentation is event generation code that encapsulates particular units of "behavior" called events. CCI predefines a primitive set of events called the *Basis Set*, which describes the lowest level of observable behavior in C. CCI's code insertion perturbs the behavior of the program in the following ways: execution speed is reduced and the insertion of temporary variables changes the program's stack allocations. CCI addresses the former by providing configuration facilities to reduce the penalty of instrumentation. Altering stack allocations limits CCI's applicability for certain kinds of

debugging problems, but was deemed acceptable in order to meet other design goals, such as portability.

CCI comprises four main phases: parsing, temporary variable creation and type propagation, configuration, and code generation. The parser builds the parse tree and symbol tables. The second phase traverses the parse tree, constructing synthesized and inherited attributes and temporary variables. The configuration phase is broken into two parts. The first part reads the configuration information before the parsing phase. Configuration controls the intrusion level of CCI and provides facilities for generating high-level information. The second part is used during the output phase. The last phase outputs a new file with instrumented source code. Each phase's design and purpose is described below.

2.1.1 Parsing the Target Program

CCI instruments the target program by parsing it using lex and yacc with a complete ANSI C grammar. CCI does not, however, correctly instrument all ANSI C programs. The exact limitations are covered in Appendix 5.1.

While parsing, CCI builds symbol tables and a parse tree that is augmented with *event nodes*, referred to as enodes. Enodes are extra parse tree nodes that serve as landmarks throughout the parse tree and are used in the subsequent phases to build, maintain, and output event generation information. Each enode defines a particular event or class of events and points to the associated subtree. Figure 2.1 shows this graphically. To insert an enode into the parse tree, a semantic action in CCI's yacc grammar builds a subtree and then attaches an enode to the top the subtree. Inserting enodes throughout the parse tree is the key to building the basis set of events. The basis set is the entire set of events which captures the observable behavior of atomic units of a C program. An atomic unit is a portion of C code that cannot be broken down any

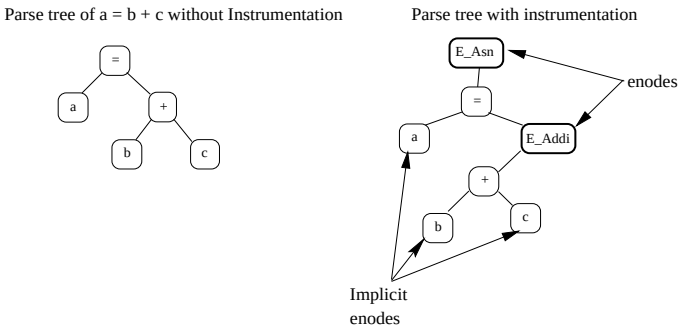


Figure 2.1: A parse tree with and without enode insertion

further without transforming that code to a lower-level language such as intermediate code. Section 2.2 covers the development and design of the basis set more thoroughly.

CCI uses symbol tables much like a compiler: to look up identifiers for type information and build output based on type information. CCI also uses type information for event generation, discussed in more detail in the following section.

2.1.2 Constructing Event Generation Information

Once the entire parse tree is built, event generation information is constructed. Consider the following C code fragment, `a = b;` Figure 2.2 shows the parse tree and inserted enode. There are four possible events for this expression: two events for the variable

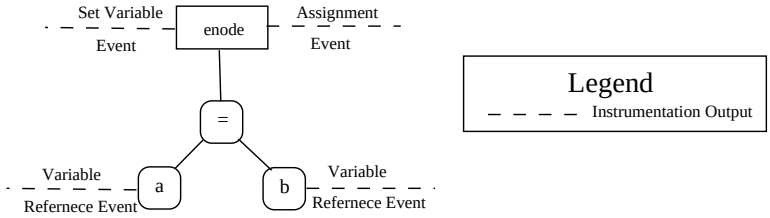


Figure 2.2: Simple expression instrumentation and its output

references `a` and `b`, another event to indicate that the variable `a` is about to receive a value and finally an event for the actual assignment. Generating four events for a simple

Table 2.1: List of macros that CCI generates.

Macro Name	Supported Types
`EVP`	`*` (pointers)
`EVI`	`int`, `short`, `char`, `long`
`EVD`	`float`, `double` `long double`
`_EV`	N/A

expression such as `a=b` is quite intrusive. One of CCI's goals is to help reduce intrusion through configuration, described in Section 2.3.

To generate an event, CCI inserts any one of four macros into the target program at the location of the event. Each macro takes two parameters, an event code and event value. The event code is an integer value that uniquely describes the particular generated event. The event value is the program value associated with that event. For example, for an integer variable reference, the event code is `E_Refi` and its event value is the address of the variable being referenced. If the variable `b` was being referenced as in the Figure 2.2, the event would look similar to `EVP(E_Refi,&b)`.

CCI does not always pass a pointer as its value: For some events a regular source language value is passed. For example, when adding two numbers `a+b`, the event code is `E_Addi`, and the event value is the result of `a+b`. If the value that is passed were always the same size as a pointer or an integer there would be no problem. However, event values of `float`, `double`, or `long double` type are larger than the `int` data type. CCI, therefore, provides separate macro invocations for various types. Table 2.1 lists the various macro names and the types they support.

One macro listed in the table, `_EV`, is a special "collapsing" macro and is defined as `_EV(a,b) b`. Figure 2.3 shows the difference between the collapsing macro and other macros. In the figure, notice the `macros` box. These are the macros that have a user-

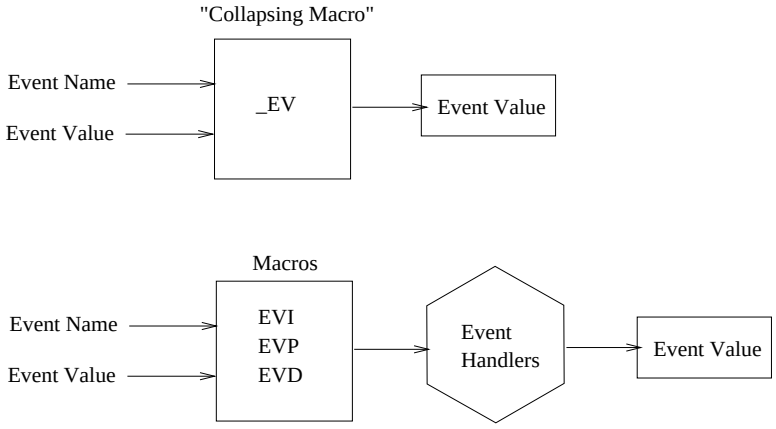


Figure 2.3: The collapsing macro's definition.

supplied definition. The code for the macro is expanded inline within the middle of an expression. Thus the macro definition must evaluate to some value; in most cases this is just the event value passed in. The collapsing macro evaluates to the event value and does not generate any event. The collapsing macro's utility is described in a later section. For simplicity `EV` is used throughout this thesis to indicate any of the user-defined macros.

As mentioned earlier, the `EV` macros are defined by the user or the monitor framework which utilizes CCI. This allows a framework to use any method to collect the information that CCI provides. However, because CCI passes address information to the event macro, the framework must have a method of making "sense" out of the addresses. As an example, the Alamo framework loads the target program into its own address space. Addresses sent by CCI are interpreted by using the symbol table information available in the target program, which is compiled with debugging on. A framework that sends event information to a file or network connection would need a different strategy for address interpretation.

While CCI does perform type checking analysis, it is not as stringent as a compiler; CCI will fail unless the target program is free of syntactic and semantic errors. Existing compilers perform these checks and the emphasis of this research is on instrumentation rather than duplicating the task of a complete compiler.

CCI uses synthesized and inherited attributes to maintain necessary type information along the parse tree. In addition CCI creates temporary variables to hold intermediate values. In a traditional compiler, for example, if there is a statement of the form: `x+y*z` the expression might be translated to the following intermediate code:

```
t1 = y * z
t2 = x + t1
```

CCI performs similar kinds of operations in order to instrument intermediate results. Temporary variable information is maintained in the enodes of the parse tree.

2.2 CCI's Basis Set

As described earlier, CCI has defined a set of events that is considered the basis set. The basis set was derived from the C grammar and forms the fundamental level of behavioral information that can be obtained using CCI. This section describes CCI's basis events in general. For a complete listing of the basis events consult Appendix 5.2.2.

The basis events are integer codes, each representing a particular low level event. The term *event name* refers to the `#define` macro name assigned to an event code. The monitor writer uses these macro definitions instead of their numerical equivalent. Some examples of event names are `E_Addi` (integer addition), `E_Pcall` (procedure call), `E_If` (if statement), etc.

CCI uses a variant of Hungarian notation for naming events to indicate rudimentary type information about the event's value. Instead of generating an event called `E_Add` for all addition events, separate event codes are used for each type of addition. For example,

given the following:

```
int a;
double c,b;
c = a + b;
```

CCI will generate an `E_Addd` event because the rules of promotion in C dictate that the result of the addition is of type `double`. Each event that contains type information has its name plus a type code mnemonic concatenated to it. The suffixes for integer, long integer, float, double, long double, character, pointer are i, li, f, d, ld, c, p respectively. To generate a long double subtraction event, for example, CCI uses the predefined `E_Subld` event name. `E_Sub` indicates a subtraction event while the `ld` suffix indicates that the event value is a long double.

Making the distinctions between types provides a finer grained event set and allows for more robust methods of selecting various kinds of events. For example, a user can configure CCI to generate events for floating point additions and nothing else. If CCI did not distinguish type, the monitor would receive an event for *every* addition event.

2.3 Configuration

2.3.1 Motivation

Using CCI without configuration produces substantial code intrusion. With configuration, the code intrusion is significantly reduced. CCI inserts macros that abstract the event generation mechanism, resulting in code explosion relative to the size of the macro definitions. Small macros increase object code size less than large macros.

Figure 2.4 shows the motivation and need for configuration. The graph demonstrates code explosion caused by using various macro definitions for one small sample program. The first bar (on the left) in the graph shows the object size of the compiled program without using CCI at all. The next bar shows the object size of the instru-

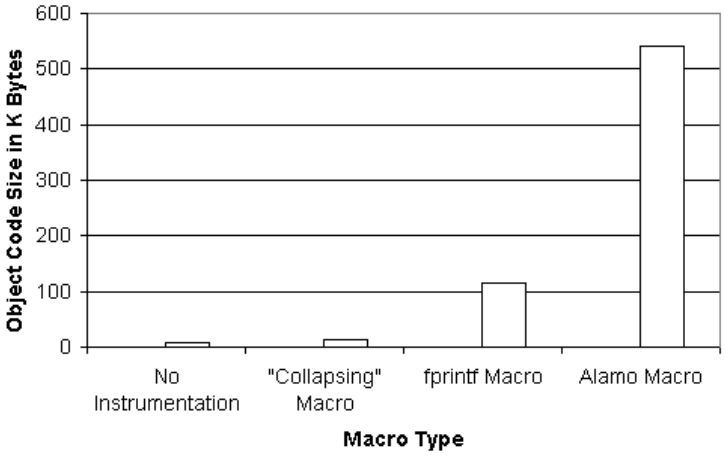


Figure 2.4: Code explosion for various types of event macros.

mented program using the collapsing macro. This macro is considered to be minimal in that it generates no event. The next bar shows the same program instrumented with a macro that only prints the event and its value to a log file. The fourth bar shows the code blow up using the Alamo event macro, which does in-line filtering of events..

In this example, the amount of code blow up using the collapsing macro for every event, is a factor of five[1]. However, the code explosion for inserting a large macro by Alamo for *every* event is a factor of approximately 80. The macros in the Alamo framework are not extremely large, but even in small programs the number of events multiplied by the macro size results in code explosion.

While Figure 2.4 is not a representative sample, it can been seen as a near best case. Code explosion means that CCI's comprehensive approach to instrumentation does not scale to handle large programs. As a result, without configuration CCI would be unusable. However, the code explosion for a collapsing macro is still approximately a factor of five. This result underlines the usefulness of configuration which reduces code

[1]The cause of this, and its solution, are discussed in the Future Work section in Chapter 5.

explosion and makes CCI a viable tool.

2.3.2 Design of Configuration

One of the design goals of configuration is to provide a paradigm of modularity to serve many different monitors in some framework. For example, one monitor may only visualize memory allocations. A configuration file could be written so that CCI will feed the monitor only what it is interested in. Another configuration file could be for a tree visualizer that expects certain events necessary for it to visualize tree structures. In this case, the user might customize a tree configuration file to map the basis events to those expected by the tree visualizer.

Configuring CCI significantly reduces the amount of intrusion and moves many run-time computations to compile-time. The configuration author configures CCI to specify the events that are needed and how to present them. Moreover, as described above, configuration authors can apply their knowledge of the source code (if available), for example structures, functions or variables of particular importance, 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 libraries.

In order to provide flexibility, CCI provides low level instrumentation which in and of itself requires monitors to do a fair amount of work to interpret and use those events. Performing all this work at run-time adds additional cost to monitoring. CCI reduces this cost while providing the ability to create high level events by using *static* filtering, the process of inserting only selected events at compile time. *Run-time* filtering on the other hand is left up to the framework that uses CCI. One example of run-time filtering is found in the Alamo framework, described earlier, which applies an event mask to the current event. If the event is in the mask, it is reported to the monitor; otherwise

execution continues.

CCI uses a simple set definition language along with additional syntax to describe what kinds of events the user is interested in. This requires that the configuration author know and understand what events are in the basis set. CCI's configuration language has facilities for selecting, refining, and composing basis events into higher level events. The configuration language design is directed by the following goals:

- It must be easy to understand for all levels of programmers.
- No programming should be needed for configuration.
- It must be flexible but should not be so general as to make it difficult to use.

By default, an empty configuration file implies that all basis events will be instrumented. If the user supplies a configuration file, then only those events selected are instrumented. A configuration file contains one or more event set *selections* and/or *definitions*, defined in the next section.

2.3.3 Event Selection

As described earlier, CCI has a basis set of approximately 160+ event codes that represent the primitive 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, the configuration author selects the desired events from CCI's basis set. To make event selection easier, CCI has many built-in sets of event codes (a complete listing of the built-in sets is provided in Appendix 5.2.2). The sets are a notational convenience. For example, `E_Add` is the set of all addition events regardless of type.

2.3.4 Event set selection and definitions

CCI's configuration files contain many event set selections and/or definitions. An event set selection is a comma separated list of event names terminated by a semicolon. The syntax for event set selection is

```
Event,...,Event;
```

Where `Event` is any one of the basis events or a *set definition*, described shortly. Say the configuration author, for example, wants to set CCI to generate only addition events. The configuration file would read:

```
E_Addi, E_Addf, E_Addc, E_Addl, E_Addd, E_Addld, E_Addp;
```

When there are many selections, the set selection may span multiple lines.

A set definition is simply a name used to denote a set. Its syntax is

```
Set_Name = (Event,...,Event);
```

where `Set_Name` is the name assigned to the set and `Event` can refer to an event name or another set name. An example follows:

```
...
E_Add = (E_Addi, E_Addf, E_Addc, E_Addl, E_Addd, E_Addld, E_Addp);
E_Math = (E_Add,E_Sub,E_Div,E_Mult,E_Mod);
```

In this example, `E_Add` is assigned to represent the addition events. `E_Math` is set to all mathematical operations involving operators. It is important to understand the distinction between set definition and selection. The former only specifies what events correspond to the set name; no events are generated from a set definition. The latter does tell CCI to generate events. The syntax for event set selection allows for selecting those events in a set. This is done by typing the set name as if it were an event name. For example,

```
E_Math;
```

CCI applies all event set selections and definitions globally to all the source files that contain those events. Support for instrumentation of specific scopes is discussed in the Future Work section of Chapter 5.

As mentioned earlier, CCI requires the framework that uses CCI to define the event invocation mechanism, passing a large portion of the performance considerations that come with instrumentation on to the framework developers. However, CCI can help performance by using static filtering and building more high level events through another special filtering technique, called *event masks*, described below.

2.3.5 Event Value Masks

CCI uses a type of event mask called a *value mask*. A value mask is a set of values that filter an event based on the event's value. Value masks in CCI are purely static or compile-time filters. For example, suppose that a configuration author wants to instrument only calls to `malloc`. If no filtering is available, the monitor would have to receive all `E_Fcall`[2] events. In addition, `E_Fcall`'s event value, the address of the function to be called, must be checked by the monitor for each `E_Fcall` event. However, by using a value mask filter applied to an `E_Fcall`'s event value, at compile time, only those `E_Fcall` events whose value is in the value mask are generated. For example, to only report calls to `malloc`, the event selection is `E_Fcall{malloc}`. In this example, CCI determines at compile-time whether the function call's identifier is `malloc`. If it is, the function call is instrumented.

Often the user is not just interested in masking an event for only one value but for many values. To monitor calls to all library memory allocation functions, the user types:

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

In this example, `E_Fcall` events are instrumented only for the functions `malloc`, `calloc`,

[2]CCI delineates between application function calls (which uses the E_Pcall event) and library function calls (which use the E_Fcall event).

and `realloc`. The problem with this is that the monitor still does not know which function was called: it could be any of the three functions. CCI solves this problem by allowing the configuration author to *refine* the basis event by substituting one or more new event codes for the filtered basis event. The syntax for an event mask is as follows:

```
Event_Name{ identifier = New_Event_Name, ...};
```

For example, events unique to the value masks given in the above example are given by:

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

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

The above examples are important in that they reinforce how masking can make one event more specific, for example, `E_Fcall` is general while `E_Malloc` is more specific. Interestingly, masking also allows for whole classes of events to be grouped together into one event. For example, one could configure CCI for all add events to generate only one event name as in the example where a call to `malloc` and `calloc` are mapped to the `E_Malloc` event.

Another example of value masks is that of variable modification trapping. To trap assignments to the variables, say `node`, `child` and `parent`, write:

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

In this example `E_Assign` is a set definition. CCI allows masks to be applied to a set of events. When applying a value mask to a set, all events in that set are given the value mask.

Value masks can also be applied to variables. Variable modification trapping is costly in many monitoring frameworks, because checking all accesses to variables (and pointers to variables) is prohibitive. Moreover, trapping variable modifications for multiple variables exacerbates the performance cost. A large number of variable accesses,

however, can be determined by CCI at compile time. Performing these computations at this stage can provide large performance gains for any framework using CCI.

CCI's value masking technique reduces the performance cost by employing static analysis. For many assignments or invocations static analysis of the target program's parse tree reveals locations where it is impossible for a particular variable to be modified. In other words for each event in CCI that can use value masks, CCI performs static analysis first to determine if an event can be filtered at compile time. This alone, eliminates an enormous number of potential event generations.

Unfortunately, static value masks are not sufficient to instrument all events based on their values because often the value changes depending on program input, etc. For example, in the previous examples, it was assumed that the value or reference is available at compile-time which is true for many typical programs; however, it is common for a program to have aliased pointers that will have potential event values that escape compile-time detection. For the same reasons, value masks on parameter variables suffer the same limitations.

2.4 Output Phase

The output phase of CCI produces instrumented source code. Output is controlled via selections in the configuration file. To properly introduce the kind of output that CCI generates, it is useful to present an example. Figure 2.5 shows an actual program that has been instrumented (on the right) and its corresponding original program (on the left). The original program's source lines are aligned with the particular lines of code that have been modified. Also, note that the instrumented program on the right has been manually formatted for this presentation: in the actual instrumented program, lines three through seven are really one line.

The lines indicated by the number one show the temporary variables created by

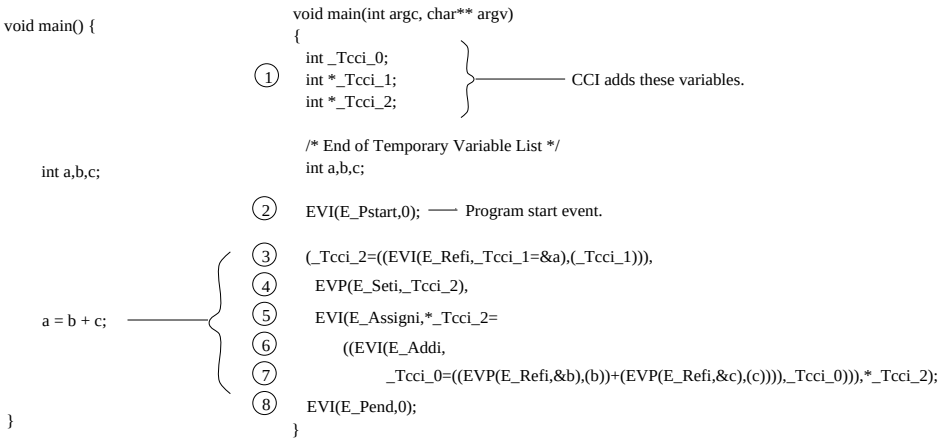


Figure 2.5: Example of a small C program's instrumentation.

CCI. Lines two and eight represent start and end of program events respectively and are found in the function `main`; if the program uses a return from `main`, CCI will also issue a `P_End` event, and if `exit()` is called instead, CCI installs an exit handler and generates the `P_End` event there as well. Lines three through seven demonstrate the technique that CCI uses. Before describing exactly how `a = b + c` was actually converted to the code on the left, it is useful to describe the basic technique used in simpler terms.

CCI uses the comma operator in C. In addition, CCI utilizes the fact that expressions delimited by commas are evaluated from left to right, and the final result of the expression is the result of the rightmost expression. For example, consider the following C code fragment:

```
a = (t2 = b + c, b = c, t2);
```

In this example, if `b` is three and `c` is two, `a`'s value is five. This is because `t2` gets the result of `b+c`. Next `b` is assigned `c`'s value, but the result of the overall expression is the rightmost expression, `t2` which is still five. Thus `a` is five. CCI uses this approach to generate events while still preserving the original behavior of the target program. CCI's basic event construction mechanism, called an *output template* is:

```
(EV( event name, temporary  = ( expression )), temporary);
```

Generally, a temporary variable is assigned to the expression, and the expression is evaluated only once. The temporary variable is then ejected as the last item in the comma delimited expression.

To reinforce exactly how CCI preserves the target program's behavior, it is useful to analyze, step by step, lines three through seven in Figure 2.5. In line three, the variable `_Tcci_1` is assigned to the address of `a` and a variable reference event is generated with the `EV(E_Refi, ... )` macro. The event value is the address of `a`. Next the temporary is ejected out of its expression which is then copied into `_Tcci_2`. [3]

Now the overall lvalue is in `_Tcci_2`. The next event generated is in line four, where the `E_Seti` event is generated. This event indicates that the variable, pointed to by the event's value, is about to be modified. Next in line seven, `_Tcci_0` receives the result of `a+b` containing the event value for the `EV(E_Addi,...)` macro. `_Tcci_0` is then ejected to the surrounding expression at line five of the figure. The result of the addition is stored into the contents of `_Tcci_2` which is `a`. Thus the result of the addition is stored in `a` when the event `E_Addi` was generated. Finally, back in line seven, `_Tcci_2` is ejected to its surrounding expression, which in this case, there is none and the expression is completed by the semicolon.

It is important to restate that this example is given without using configuration and results in a large number of events for only one simple statement. This example shows the amount of detail that is possible from CCI. But the user usually does not need this level of detail. The user configures CCI both to specify higher level events and to avoid instrumenting events that are not of interest, potentially reducing the overall number of events.

[3] To the astute reader, it may seem that this is redundant, and it is in this particular case. The _Tcci_2 variable is supposed to capture the entire lvalue expression which in this case is `a`. This additional temporary only proves useful when the lvalue is an expression. For example, `*(a+1) = c`.

Chapter 3

Implementation of CCI

CCI consists of approximately 15,000 lines of source code and 25 modules. This chapter covers implementation with each section covering a phase of the design discussed in Chapter 2. The order of discussion, however, is slightly different. Configuration's implementation is covered first, followed by instrumentation and finally output, representing the order in which CCI actually executes.

3.1 Configuration

When CCI first executes, it parses the configuration file, filling up an event selection table. The event selection table is used by the Output Phase to produce instrumented code. This section first describes how the basis set was constructed and argues for its completeness, then discusses how CCI parses the configuration file and builds the event selection table. Finally configuration's internal representation is presented.

3.1.1 Basis Set Construction

The basis set is a static set of events, developed by analyzing the C grammar. Events are associated with those grammar rules that result in generated code, since generated code is required to produce most observable program behavior. The basis set was constructed by examining the C grammar, rule by rule, and building a list of events. Events should be generated when the target program performs a variable access, a typecast, a procedure call, or a binary operation, etc. The fact that a typecast should generate an event may surprise the reader; however, typecasting does provide semantic information contributing to program understanding.

The basis set is complete. No offer of a formal proof is made; instead a practical case is presented: every rule of the C grammar was analyzed. If any rule represented any possible behavior an event was constructed for it.

Each event's various types were determined during implementation because several events can be associated with a particular grammar rule. For example, for addition in C, the event's value will be a numeric result of two expressions being added together. The event's type and each expression's type depend on the promotion rules of C. Because CCI differentiates events based on type, there are six possible addition events (one for each data type).

Some events, however, do not have multiple types associated with them. The modulus operator `%` can only be used with integer operands. Thus there need only be one event named `E_Mod`. Most control-flow events also do not have type variations.

Once all the events were determined, two files were created, `evmap.h` and `evcodes.h`. `evmap.h` is used by CCI to initialize the Event Dictionary (see next section) with the basis set. The second file, `evcodes.h`, is a list of `#define`s consisting of event names and a number. For example, `#define E_Addi 100`.

3.1.2 Parsing the Configuration File

For configuration, CCI builds an object called the Configuration Manager (CM) consisting of two layers, shown in Figure 3.1. The first layer is the Event Dictionary Layer (EDL), the second is the Event Selection Layer. The EDL stores the basis set and any

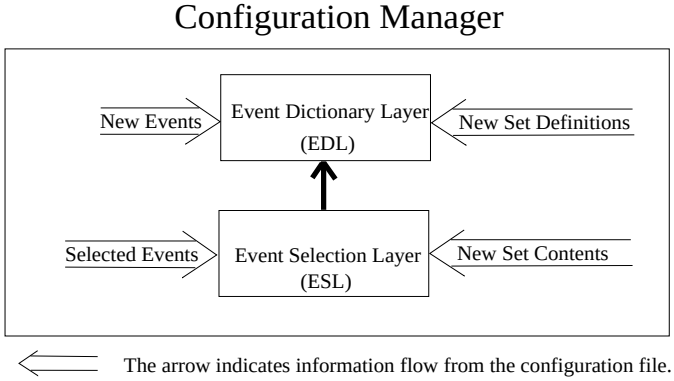


Figure 3.1: The layers of the CM and its information flow from the configuration file.

new event names or set names and also assigns an event code to the event or set (sets make use of the event code differently and are described in more detail later). Figure 3.1 shows what layer is responsible for maintaining the various kinds of information delivered by the configuration file. The large arrows pointing into the EDL, representing information flow from the configuration file, indicate where the sets and new events, along with the basis set, are stored. The bold arrow from the ESL to the EDL indicates that all the event selections are really pointers into the EDL. To summarize: recall that in the mask example `E_Fcall{malloc=E_Malloc}` the new event name, `E_Malloc`, is stored in the EDL along with a new event code. However, the actual selection of the event `E_Fcall` and the information containing the value mask is contained in the ESL.

Specifically, the ESL consists of a table of event selections, masks, and set definitions. In future work, the ESL will maintain multiple tables, one for each scope. For more information about applying configuration with respect to scope, see Chapter 5.

The implementation of these tables is discussed in the following section. The rest of this section describes how the grammar's yacc action statements "fill up" the EDL and ESL.

When an event is selected via configuration, a selection entry is place into the ESL. For example, in Figure 3.2, `E_Fcall` is selected and masked using the refinement, `E_Fcall{malloc = E_Malloc}`.

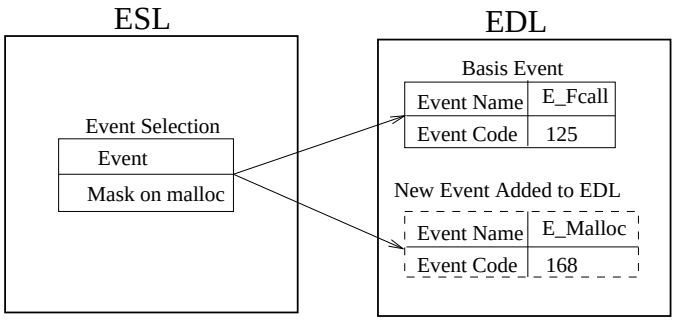


Figure 3.2: How the EDL and ESL work together to store event selections.

First the `E_Fcall` event is selected into the ESL. Because the information about `E_Fcall` is already in the EDL, the event selection points to the EDL's information. Next a new event is added to the EDL, `E_Malloc`. The mask *value* is stored in the ESL as a part of the event selection's information. When new events are added they must receive a new event code which must be unique. The new event is visible for the rest of the configuration file. For example, if the configuration file follows with the line `E_Fcall{calloc = E_Malloc}` the event `E_Malloc` is to be generated if either `calloc` or `malloc` is called. The resulting structure is shown in Figure 3.3. The figure also shows how CCI's EDL and ESL work together to allow two or more event values to map to one new event name.

Another type of configuration feature is set construction. For event set construction, the grammar is given by:

```
EventSet: Identifier ASN EventListSet
```

A set is constructed using this rule which expects an identifier followed by the equal

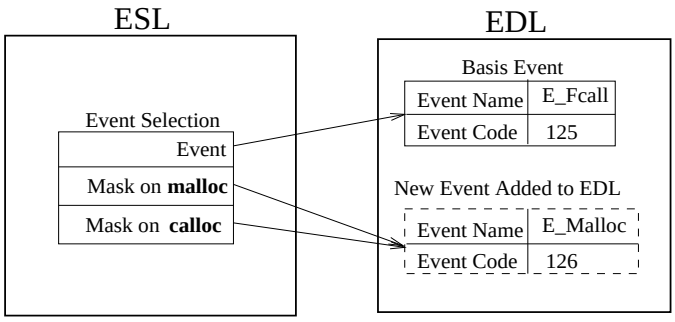


Figure 3.3: Multiple mappings of mask values to one event.

sign followed by `EventListSet` which is list of events separated by commas and enclosed within parentheses.

The actions associated with this rule create an event set object and enter it into the ESL. Event names not in the `EventListSet` are not selected in the ESL but instead are added to the current event set. Figure 3.4 shows how the EDL and ESL work together to manage set definitions. This figure shows how the set definition

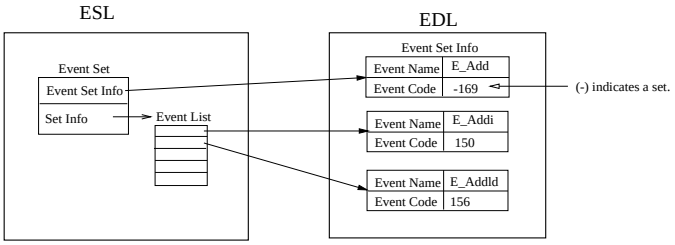


Figure 3.4: How the EDL and ESL store set definitions.

`E_Add(E_Addc,E_Addi,E_Addf,E_Addd,E_Addld)` is entered into the event table. Even though `E_Add` is not an event, it represents a set of events and could be used later in the configuration file to select all the events that `E_Add` represents; therefore, it is added to the EDL. The event code is negative, indicating the entry is an event set and not an event.

The following sections describe the implementation of the EDL and ESL.

3.1.3 Implementation of the Configuration Manager

The CM's infrastructure was presented in Figure 3.1 in a simplified form. This section discusses the layers represented in the figure in more detail.

The first layer, the EDL, consists of two hash tables and is shown in Figure 3.5. One is indexed using the event name and the other indexed using the event code. This allows lookups using either of the two pieces of information. The basic event data structure, also shown in Figure 3.5, called `EvInfo`, contains the essential information for any event: the name and its event code. The EDL consists of pointers to `EvInfo` nodes. The actual name of the event is stored in the `EventMap` array, an array of event name strings exported in `evmap.h`.

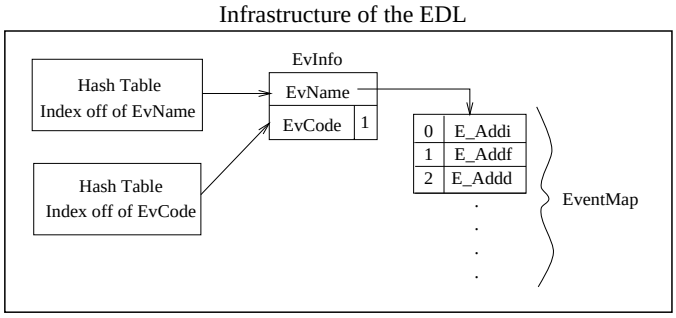


Figure 3.5: Event Dictionary Structure.

When CCI first starts, the basis set is initialized by iterating through the `EventMap` array, constructing `EvInfo` objects for each event and inserting them into both hash tables. Then during parsing of the configuration file, it is trivial to determine if a particular identifier is an event or not. The identifier matching action, in the configuration file's lexical analyzer, looks up the current token in the EDL; if the token is in the table, it is an event.

Once the EDL is constructed, the configuration file is parsed. As described in the previous section, the grammar parses the configuration file filling up the ESL with

selected events and possibly their masks, and any set definitions. Figure 3.6 shows the ESL's structure.

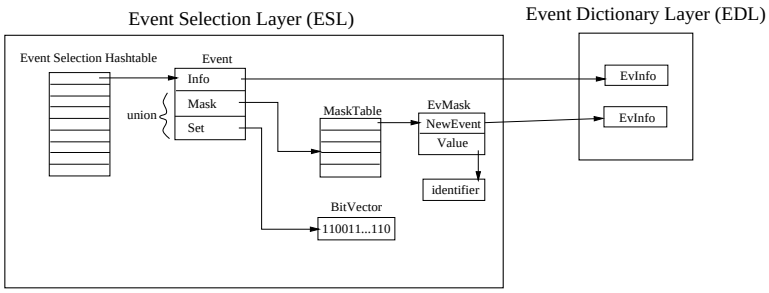


Figure 3.6: Event Selection Layer's structure.

The ESL, is a hash table which points to an `Event` object. The `Event` object, shown in Figure 3.6, contains a pointer to an `EvInfo` object. Recall that `EvInfo` holds the event's name and its value. In addition, `Event` contains a union of two pointers, pointing to either a `MaskTable` or a `BitVector` object. These two objects are described later. For now, consider the following configuration statement:

```
E_Refi, E_Seti;
```

this line configures CCI to generate events for integer variable references and integer storage events. For events such as `E_Refi`, the event is looked up in the EDL to obtain a pointer to the `EvInfo` object. This pointer is then passed to a function which enters the event into the ESL. This is repeated for all events in the event selection list. This structure maintains specific mask or set information and points to the appropriate `EvInfo` object found in the EDL.

In order to implement a flexible masking mechanism where multiple masks pertain to the same event, each potentially creating a new event name, a `MaskTable` is constructed for each selected event that has a mask applied to it. The `MaskTable` is a hash table, representing a set, which contains `EvMaskElem` objects. These objects represent an

element of the mask set and point to an `EvInfo` object representing the new mask event. `EvMaskElem` also contains a pointer to `char` representing the value mask information. For example, Figure 3.7 shows the mask structure using a previous mask example:

```
E_Fcall{malloc=E_Malloc,calloc=E_Calloc,...};
```

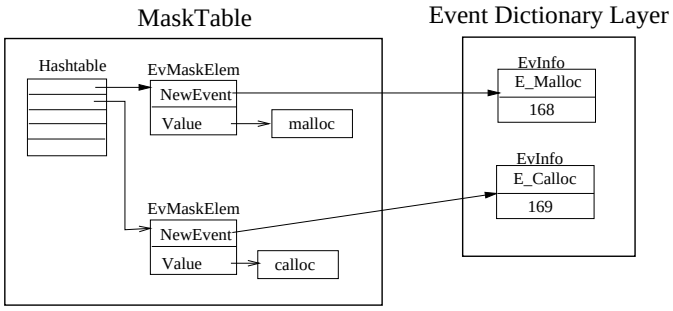


Figure 3.7: The MaskTable and the EvMaskElem object.

Part of the union of the `Event` structure is the `EvSet` which represents a set definition. Recall that CCI places the event set's name and a code, which is negative, in the EDL. The ESL, however, stores the set's definition. Figure 3.8 shows how CCI manages sets. One component of the `Event` structure is a `BitVector` which is a dynamic array of bytes. Each bit position of the bit vector represents an event or an event set via the absolute value of the event code (the absolute value is necessary because events sets have negative values for their event codes). If the bit is on, then the event belongs to the set; otherwise, it is not in the set. Using a `BitVector` makes a union of sets a simple operation of or-ing two or more bit vectors together.

When a set is selected the CM must select all the events within that set. In addition, if the set is made up of other sets, those events in subordinate sets must also be selected. Using the above implementation this is a straight forward process. For example if the configuration files selects the `E_Add` set, where `E_Add` is the set of all basis addition events, the CM iterates through the bit vector. For each bit that is set the following

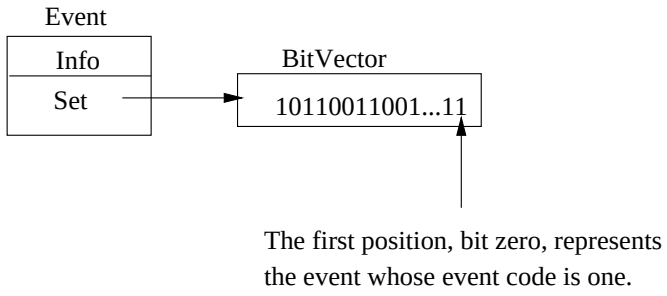


Figure 3.8: EvSet object.

algorithm is used:

1. Look up the event in the EDL via the event code which corresponds to the current bit position in the bit vector.

2. If the event is a set (where the event code is negative) then recursively apply algorithm.

3. If the event is not a set then add an event selection to the ESL.

Once all selected events are entered into the ESL, input and parsing target program source code begins.

3.2 Parsing the Target Program

Before the target program can be parsed, it must be preprocessed by expanding macros, `#define`s and `#include`s, etc. CCI uses the gcc compiler with the `-E` option as its preprocessor.

The output from the preprocessor is fed in to the lexical analyzer. The lexical analyzer tokenizes the input stream passing the result on to yacc. As mentioned in the previous chapter, CCI uses a complete C grammar and builds a parse tree and symbol tables.

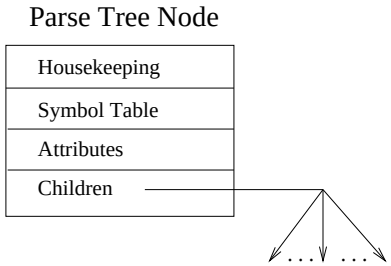


Figure 3.9: Parse Tree Node Structure

In a some compilers, type propagation may occur at the same time the parse tree is built. However, in CCI, type propagation is performed during a second pass and is covered in Section 3.3.

3.2.1 Parse Tree Construction

CCI builds a parse tree bottom up from leaves to the root of the tree. This is a natural way to build a tree when using yacc. The parse tree nodes contain the necessary information to traverse the tree, facilitate easy symbol table access, and store synthesized and inherited attributes. Figure 3.9 shows the general layout of a parse tree node. The housekeeping section stores various pieces of information pertaining to the parse tree, such as which rule the node belongs to, what line number and filename, pretty printing information, etc. The symbol table section points to the symbol table belonging to the node's scope. Attributes are the synthesized and inherited attributes which help build up type information. Finally "children" point to other parse tree nodes.

To build the parse tree, CCI uses yacc. For each yacc rule, the action section builds a parse tree node, attaching to previously created subtrees. In addition, if that rule is also the location for an enode, the enode is inserted. For example, consider the following yacc fragment:

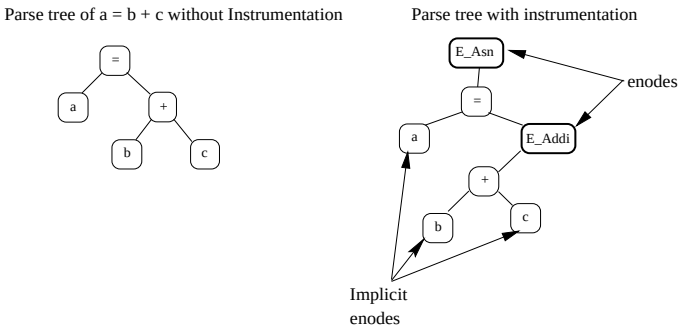


Figure 3.10: Parse tree construction and insertion of enodes.

```
additive_expression:
   multiplicative_expression
   | additive_expression PLUS multiplicative_expression
     { $$ = ENode(en_op_exp,binary(ADDITIVE_EXPRESSION + 2, PLUS,$1,$3)); }
   ...
   ;
```

The `additive_expression` grammar rule's action constructs a syntax tree for addition and subtraction (subtraction not shown) expressions. `binary` takes as its parameters, the rule to which the parse node belongs, pretty printing information, and two more parameters `$1` and `$3`. These parameters are subtrees created from previous rule reductions.

In addition, because this rule defines program behavior, an enode is attached to the top of the tree. This is done by calling `Enode()`, which takes two parameters. The first parameter is the event name or event class which describes a particular group of events. The second parameter is the subtree constructed by the call to `binary()`. Figure 3.10 shows the construction graphically. CCI must be able to instrument identifier loads and/or stores. If CCI were to attach enodes for every identifier, this would entail an unnecessary memory cost. Instead, as in Figure 3.10, CCI uses the leaves themselves to serve as implicit enodes.

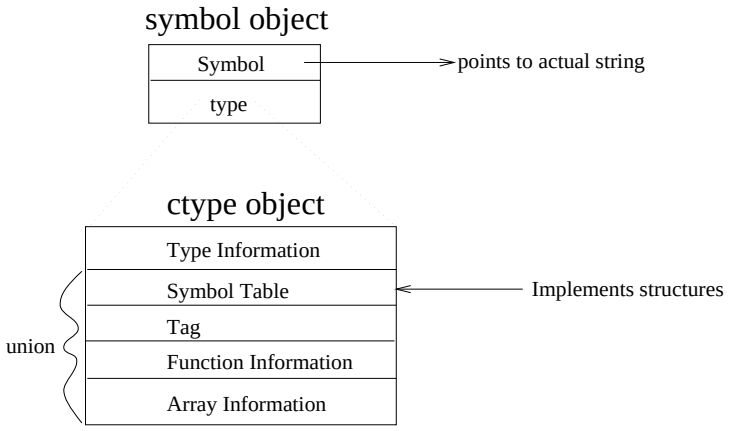


Figure 3.11: Symbol object and its ctype object

3.2.2 Determining an Event Value's Type

Many events have type information associated with the event code, but because CCI defers type propagation until later, there is no way to know the proper event code at the time the enode is created. For example, the code `a = b + c` in this form could result in an `E_Addi` event or an `E_Addf` event. This problem is solved in two steps. When `Enode()` is called to create an event of unknown type it just assumes the event will be of integer type (in this case `E_Addi`). Later, when the type information is available a function is called to update the `enode`'s event code with the appropriate type.

As mentioned earlier, CCI builds the symbol tables as it is parsing the grammar. The implementation of the symbol table is typical of any compiler and is not discussed in much detail here. However, the symbol table maintains identifiers and their types using a `Symbol` object, shown in Figure 3.11. The type is created in the grammar and is stored using a `ctype` object. The `ctype` object is also shown in Figure 3.11. An interesting aspect of CCI that is not typical of a compiler is that CCI must be able to traverse any `ctype` object and produce legal C declarations from it. This is necessary in order to generate declarations for temporary varaibles and is described in the next section.

3.3 Instrumentation

The actual process of instrumentation is multi-phased. During parsing, enodes are inserted into the parse tree. Once the parse tree is fully constructed, the second phase performs an in-order traversal of the parse tree. As the recursion moves down the parse tree inherited attributes are propagated down, and as the recursion unwinds, synthesized attributes are passed up. The inherited attributes are lvalue and rvalue information, and the synthesized attributes are type information, temporary variable generation, and identifiers that could contribute to a value mask.

The following section motivates the use of temporary variables and describes how they are manipulated and used. The next section covers how and when they are created. The final section covers how identifiers are moved up the parse tree for use as potential value masks.

3.3.1 Motivation for Temporary Variables

As mentioned earlier, temporary variables are needed to hold intermediate values of expressions. This is necessary to avoid altering the semantics of an expression. For example, if there is the following code fragment: `a = b + foo();` CCI must instrument this line without executing `foo()` more than once. Thus CCI uses the following recursive pseudo code:

For each node in an inorder traversal of the parse tree do

1. For every rvalue expression that is not an identifier generate a temporary variable.
2. For every lvalue expression generate a temporary variable.

For this example, CCI would generate, in pseudo code:

```
((T2 = &a),*T2 = (T = b + foo(),T),T2);
```

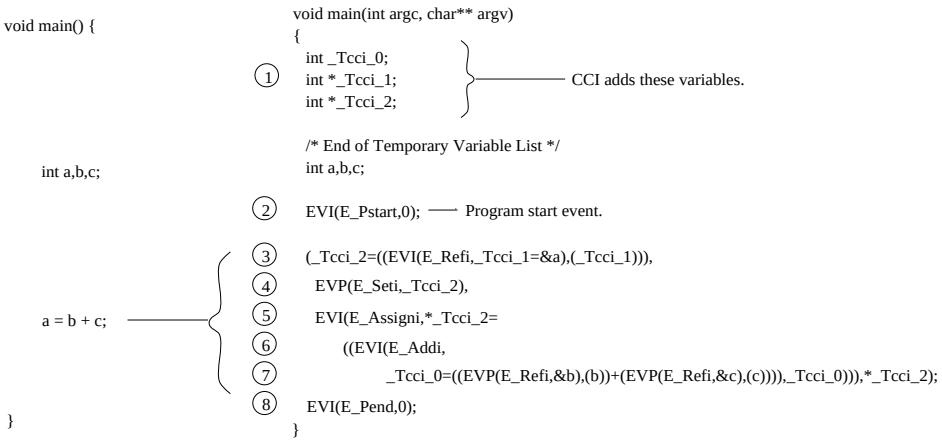


Figure 3.12: An example of inserting temporary variables.

This example has event generation code removed. Notice that the temporary `T` "captures" the results of the expression `b + foo()`. Then `T` is ejected out to be assigned to `*T2`. This, in effect, has assigned the sum to the variable `a`.

In addition to using the temporaries, CCI must also declare them in the source code so that it will compile correctly. Two declaration methods are possible. One method is to declare all the temporary variables globally. However, this method is not recursion or thread safe. Therefore, CCI declares the temporary variables inside the function in which they are used. An example of this is given in Figure 3.12 (reprinted from Chapter 2 for convenience). Notice that the temporary variables do not necessarily have the same type as the variables in the original source program. The reason for this is that sometimes CCI needs to report addresses and therefore needs to create pointers to the identifiers.

The temporary variables are represented in CCI using the `Symbol` structure, the same as all other identifiers. Recall that this structure contains a pointer to the identifier name and a pointer to the `ctype`. When CCI outputs the target program, it must output the temporary variable declarations. To do this, CCI attaches a list of variables to an `enode` that marks the beginning of the declaration section. Figure 3.13 depicts this implementation. The `enode`, in the figure, is inserted when the parse tree is built. In

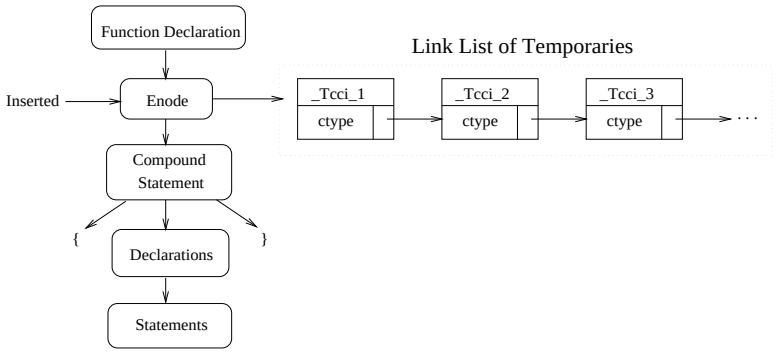


Figure 3.13: How temporary variable declarations are inserted into the parse tree.

addition, this particular `enode` does not produce any event generation code, but serves as a "landmark" in the source program's parse tree. Once instrumentation is complete, the output phase stops at this enode and outputs the link list in the form of C variable declaration code.

3.3.2 Attribute Propagation and Temporary Variable Usage

The other phases up to this point described setting up the infrastructure needed for instrumentation. This section describes how the temporary variables are created and used to render instrumented output.

CCI checks each node in the parse tree, with an inorder search, to see if the node is a leaf node or an enode. Recall that a leaf node is an implicit enode. If the node is an enode, CCI inspects the enode's event code and calls a routine to handle instrumenting that event, called an *event handler*.

All recursive algorithms need a base case which will "unwind" the recursion. Leaf nodes serve that purpose and are the key to type propagation and temporary variable creation. At these nodes, which contain identifiers, CCI determines what temporary variables to create for instrumentation[1].

[1]After successfully implementing CCI, it became apparent that few identifiers ever need temporary

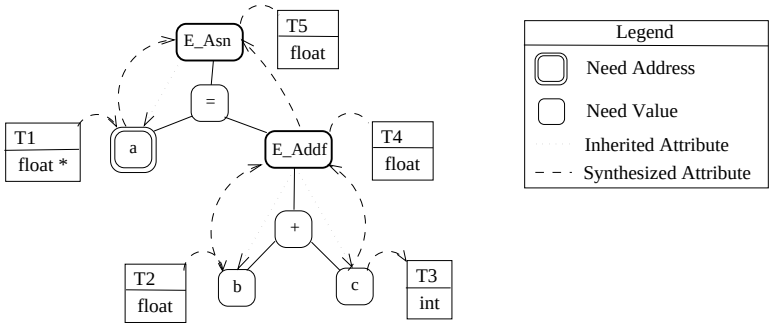


Figure 3.14: Flow of inherited and synthesized attributes and temporary variable creation.

A temporary variable is created which reflects the kind of type information needed. This is determined by an inherited attribute, `NeedValue`, which specifies if this node is supposed to synthesize an address or a value. Figure 3.14 shows the flow of the inherited attributes using the dotted arrows. The synthesized attributes are shown using a dashed line. The double circle around `a` indicates that its inherited attribute expects an address. Because it is an lvalue, the temporary variable is created as a pointer to the identifier. Every temporary variable is stored with its enode for later use during the output phase. Once the temporaries are bound to the enodes the type information is propagated up to next enode which in turn determine the types of new temporary variables. This process repeats up the entire tree.

Once all temporary variables have been created and their types determined, the final instrumentation phase begins the output phase.

3.3.3 Value Mask Propagation

One of the important features of configuration is the use of value masks. Recall that a value mask is applied to an event to determine if the value of that event will equal the

variable assignments. See the Section 5.2 for a more thorough discussion.

mask value. The value of the event, at compile time, can only be an identifier. The problem is that an enode may not directly point to the corresponding identifier node. In fact, it may be many nodes away from the actual enode.

Because all identifiers are considered potential value masks, they are propagated up the parse tree along with the other synthesized attributes. Parent nodes that only have one child may synthesize the value mask attribute for use by their event node, if one is present; if not, the attribute is synthesized anyway. When the parse tree node is a parent of two or more children then it represents an operator on or expression of the identifiers and, therefore, their event is not applicable to value masks.

CCI makes no other attempt at value mask assignment. That is, it does not perform any compiler-like data flow analysis to determine whether the pointer is really referencing some other variable that is being value masked.

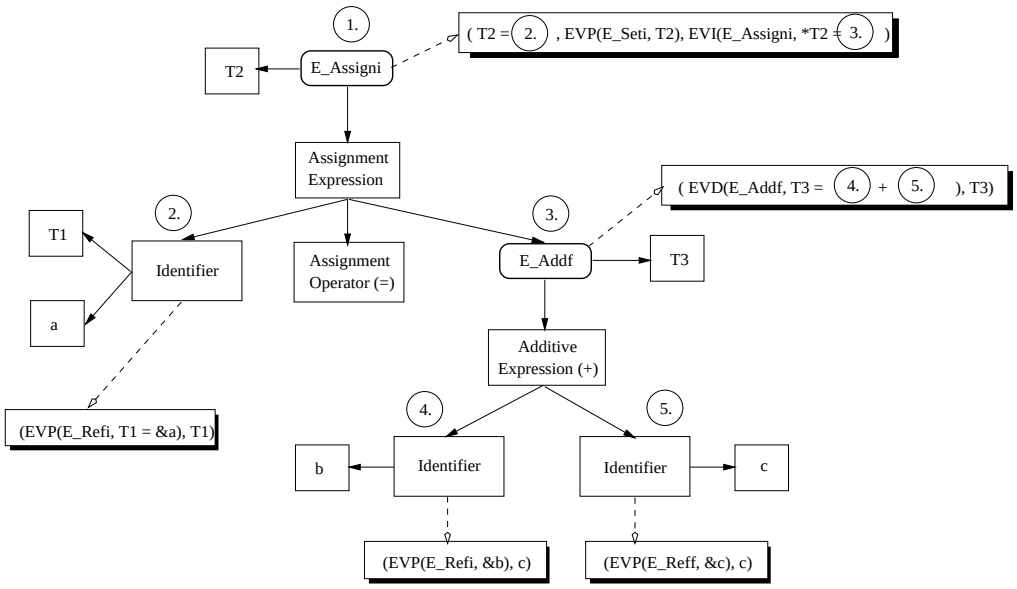
3.4 Output Phase

CCI writes the new source code to a file by traversing the parse tree. Like the instrumentation phase, the output phase must also check for enodes. For each enode found, CCI looks up its event code and calls an *output handler*. Each output handler is set up to analyze all the attributes of an enode and determine how to output the instrumented code.

3.4.1 Output Templates

For each expression, CCI uses an output template. The output template is generally of the form

(EV(*event name*,T= (*output of child expression*)), T_a)



Goal: (T2 = (EVP(E_Refi, T1 = &a), T1),(EVP (E_Seti, T2), (EVI (E_Assigni, *T2 = (EVI(E_Addf, T3 = ((EVP(E_Refi, &b), b) + (EVP(E_Reff, &c), c)), T3)))

Figure 3.15: A complete example of CCI-rendered output of the expression a = b+c.

`EV` is really any one of a set of macros, and is also the key to providing simple configurability. *Event name* is the `#define` reference. The bold `T` is a temporary variable which takes on the output of the expression below the current node. Finally, $\mathtt{T}_a$ represents the temporary variable, which might be augmented, in one of two forms: either `T` or `*T`. $\mathtt{T}_a$ depends on the inherited attributes and the type of the temporary variable, the output handler determines if it needs to eject the temporary variable `T` or the contents of `T`.

Figure 3.15 shows how a simple expression `a = b + c` is instrumented and rendered. At the bottom of the figure is the complete instrumented form assuming all events are generated. Figure 3.15 is formidable, but if each of the various parts are described then understanding how CCI actually renders its output is possible. The figure represents an instrumented parse tree for the above expression. The shadow boxes represent output templates and are not stored at the enodes, but the output handlers know their format. The dashed arrow denotes that the output handler is associated with

that particular enode. Each output template contains a circled number which also labels the enode responsible for its output. These numbers indicate the order of the recursive traversals of the tree, each node producing its own output. The result is denoted by the "goal" at the bottom of the figure.

CCI's major contribution to instrumentation, however, is configuration. Section 3.1 discussed the implementation of configuration, however, the next section discusses how the output phase uses the Configuration Manager to render configured instrumented output.

3.4.2 Controlling Instrumentation Via Configuration

CCI requires that a set of macros be defined that are used to generate events. The collapsing macro, however, is automatically defined by CCI. Its definition, as a reminder, is `#define _EV(a,b) (b)`. The essence of this macro is that no event generation will occur and the expression `(b)` will be evaluated. To implement configuration, CCI uses this technique and is described in more detail below.

Recall the output template in the previous section. CCI must determine the exact `EV` macro name for the event. For example, notice that in Figure 3.15 two of the macros' names are `EVD` because, as a pedagogical device, their values are of type `float`; whereas the others' are `EVI`. The various names were discussed in Chapter 2, Table 2.1.

To realize the benefits of configuration, CCI changes the name of the macro for any unwanted event to `_EV`. As an example, say for the expression `a = b + c`, that only addition events are of interest; that is the configuration file only contains `E_Add` as its selection. Then the only change in the output is the macro names. For example, only the macro for the `E_Addf` event would remain the same, the others would change to begin with "`_EV(E_NOP`."

To determine if an event should be generated, CCI calls `Macro()` which takes two

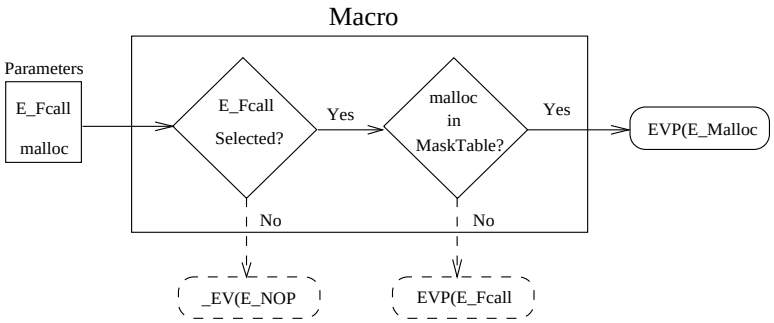


Figure 3.16: Macro function's algorithm.

parameters and returns a string of the form `EV?(eventname` where `?` is the letter of the actual macro. The parameters to `Macro` are the event name and the value mask. Figure 3.16 shows the `Macro`'s algorithm. `Macro` calls the Configuration Manager to look up the event name in the `ESL`. If the event is selected the CM also checks the event's `MaskTable` for the value mask. If it is in the `MaskTable` the new event name is used instead of the one passed in. The solid arrows in Figure 3.16 show the path taken given the parameters.

If the event is not in the ESL, however, `Macro` produces the macro output "`_EV(E_NOP`", indicated in Figure 3.16 by the dashed arrow and output box. This technique prevents any unwanted events from calling the macros defined by the framework developer, in effect reducing code intrusion and explosion.

This implementation does have some drawbacks. The most salient is that many expressions are still transformed using the output template and many unnecessary temporary variables are introduced. However, these problems notwithstanding, CCI still considerably reduces code intrusion and blow up. In addition, implementing configuration in this manner was simpler and allows for users to begin immediately using CCI in a realistic environment. Chapter 5, Section 5.2 discusses improvements to CCI's configuration methods.

Chapter 4

Performance

The goal of this chapter is to illustrate the effects of configuration by measuring the run-time execution speed and code size of instrumented programs. Because CCI instruments a target program in order to reveal particular behaviors of a program, the execution speed of the instrumented program depends on both the program's behavior and the kind of events the particular monitor needs, which together determine the actual number of events it generates. If a monitor wants every event, code explosion and greatly reduced execution speed is unavoidable. Fortunately, this is seldom the case. Typically a monitor or a set of monitors only need a small subset of the basis set events. The events needed by a set of monitors is the union of the events needed by each monitor.

The instrumented program's code size is increased by the additional code that CCI inserts and by the macro size employed by the framework. When comparing execution speed of an original program to its instrumented version, the difference is affected by the instrumentation cost, the run-time cost of each generated event called *framework overhead*, the cost of monitoring the event, and the total number of events generated.

The cost of instrumentation is the execution-time associated with the additional code used in CCI to hold intermediate expressions. Framework overhead is associated

with the execution time of handling any event regardless of whether it is used by a particular monitor. The cost of monitoring an event is the time spent on a monitor's analysis and presentation of that event. These times vary from one event to another because the framework may perform different calculations for different events. The total number of events generated depends on the behavior of the program and how CCI is configured. Most of the frameworks used in the performance measurements employ event macros that are small in code size and have low monitoring time cost. Each framework's overhead cost was kept constant throughout the trial runs.

Performance measurements were calculated using four frameworks that use CCI. A framework is a set of macro definitions for CCI's instrumentation macros. In order to more clearly measure the effects of configuration, event counters were used for all macros, except for the log file framework. The four frameworks and examples of their macro definitions are shown in Table 4.1.

Table 4.1: Frameworks and their macro definitions used in performance analysis.

Framework	Macro `#define EV(a,b)`	Description
Inline	`_ccievents[(a)]++`	Increment counter for specific event
Procedure	`_cciCount(a)`	Calls a function `_cciCount()` which performs the same function as the Inline framework
Logfile	`fprintf(evf,\"%-8s %8d\n\",EventMap[(a)])`	Writes to a logfile the name of the event and its value
Alamo	Because Alamo's macro is large, pseudo code is provided below: is event in event mask? yes - Perform context switch to monitor continue execution	

The Inline framework's macros are the smallest in code size and run-time cost. The Procedure framework defines the macros so that they call a function which updates an array of event frequencies. These two frameworks are fairly low cost and have a

constant, low framework overhead and monitoring cost. The Logfile framework records all the events and their values to a log file[1].

The Alamo framework, described in Chapter 1 employs macros that perform inlined, run-time event filtering. If the event passes through the filter, the macros perform a light-weight context switch to the execution monitor and another back to the target program. The execution monitor used by Alamo in these experiments maintains the total count of all generated events and performed no run-time filtering. To be fair, it should be noted that using no run-time filtering weakens Alamo's run-time performance because without the run-time filtering Alamo will perform the context-switch for *every* event that CCI generates. Because the purpose of these measurements is to evaluate CCI's configurability and not Alamo's performance, Alamo's run-time mask was set to filter **no** events. Section 4.3 evaluates how CCI's configurability contributes to Alamo's run-time filtering in reducing execution time.

In these experiments, the goal is to measure the cost of instrumentation and how configurability can reduce it; therefore the monitoring run-time cost is approximately constant and is also considered to be the best case for each framework. One can expect slower execution times as the monitoring run-time cost increases, but as shown in this chapter, configuration provides substantial benefits independent of what the monitor does with the events.

The programs that were used for performance measurements are small but their iterative nature results in many events during execution. The performance measurements given in this chapter are meant to show the value of configuration for each program's execution with a given framework and nothing more. Each program's behavior depends on their inputs and how CCI is configured. The goal of the measurements is to give the reader a feel for the necessity and importance of configuring CCI for only events that are

[1]Actually the output for the experiments is routed to /dev/null because a small program still can produce over a 100MB trace file making it impractical to test.

of interest.

4.1 Measurements

4.1.1 Protocol

Three programs were used to measure performance. Each program was compiled using `gcc -g` , linking in the math library and was tested on a Sun UltraSPARC running at 167MHz with no other user processes running. Table 4.2 lists the programs used, their uninstrumented object size and execution speed without instrumentation along with a brief description of what the program does and their inputs.

Table 4.2: Programs use for performance measurements

Name	Description	Code Size	Speed (in seconds)
concord.c	Prints concordance of words in alphabetical order. Its input was CCI's y.tab.c file	10K	0.33
life.c	Game of life on a 20x20 board with one spinner and 30 iterations	9K	0.24
laplace.c	Solution to Laplace's equation at a point using Monte Carlo method with 1000 approximations	7K	0.63

For each program, several configuration files were used to instrument the program for different behaviors. Each of these configurations were tested against the four frameworks, introduced in the previous section. The configurations used to measure performance are given in Table 4.3. To reference the particular configurations throughout this chapter, the configuration filename will be used.

For each program, each configuration was compiled with each framework, the

Table 4.3: Description of the configuration files used in performance measurements

Name	Description
all.cfg	Every event was selected for instrumentation
stores.cfg	All assignments and variable accesses were instrumented
structs.cfg	Only structure access and assignments were instrumented
math.cfg	All mathematical binary operators were instrumented
calls.cfg	All function calls were instrumented
fcalls.cfg	Only library calls, for example `fprintf()`, were instrumented

program was executed five times and a trimmed mean wall-clock execution time was recorded. Because the Alamo framework spends time in the operating system kernel due to context-switching, wall-clock execution time was used instead of real cpu time. The program's code size was also recorded. The results are discussed below.

4.1.2 Code Explosion Measurements

Figure 4.1 shows the increased object code size for the Alamo framework, for each configuration and program. The instrumented program name is the independent axis. In Figure 4.1 the all.cfg bar represents code explosion yielded by configuring CCI to generate all events using this framework. Table 4.2 shows that `concord.c` compiled, without instrumentation, yields a 10K program. Instrumenting `concord`, `life`, and `laplace` using the Alamo framework and configuring CCI to generate all events significantly increases object code size. Figure 4.1 shows that when all events are instrumented the code size increases by a factor of approximately 60, 40, and 28 respectively. When the programs were instrumented with other configurations their code explosion was reduced significantly. The worst-case improvement, in Figure 4.1 is for the stores.cfg configuration,

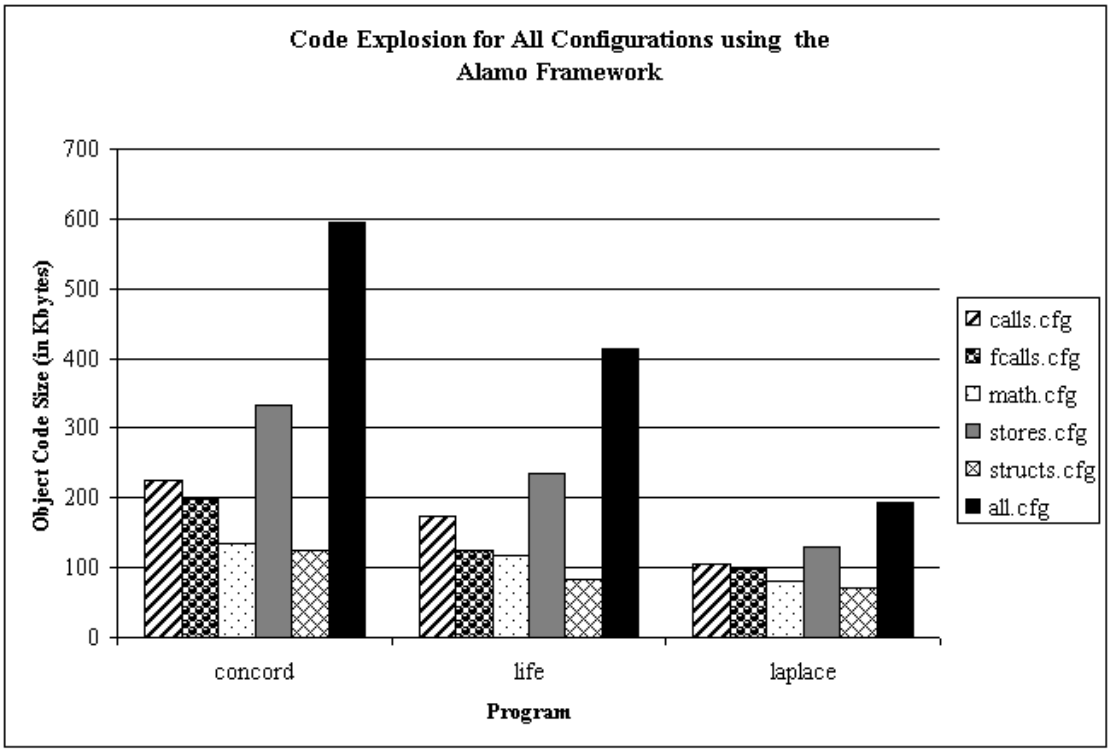


Figure 4.1: Code Explosion Measurements for the Alamo Framework

which had a reduced code explosion, for each program by a factor 30, 25, 18 respectively. The best-case improvement, for these programs was the math.cfg configuration which yielded code explosion factors of 15, 12, and 11. These numbers shows that configuring CCI to instrument a subset of the basis events can reduce code explosion from 1/2 to 1/4 of the code explosion associated with instrumenting for all events. Since CCI introduces a factor of six increase in code size even when no events are configured, code bloat can be reduced by optimizing the output that CCI generates. This issue is addressed in Chapter 5.

The code explosion for the Alamo framework is due to the macro size which is much larger than the other frameworks' macros. However, code size increases are substantial for all frameworks. For example, Figure 4.4 shows that when using `all.cfg` with the

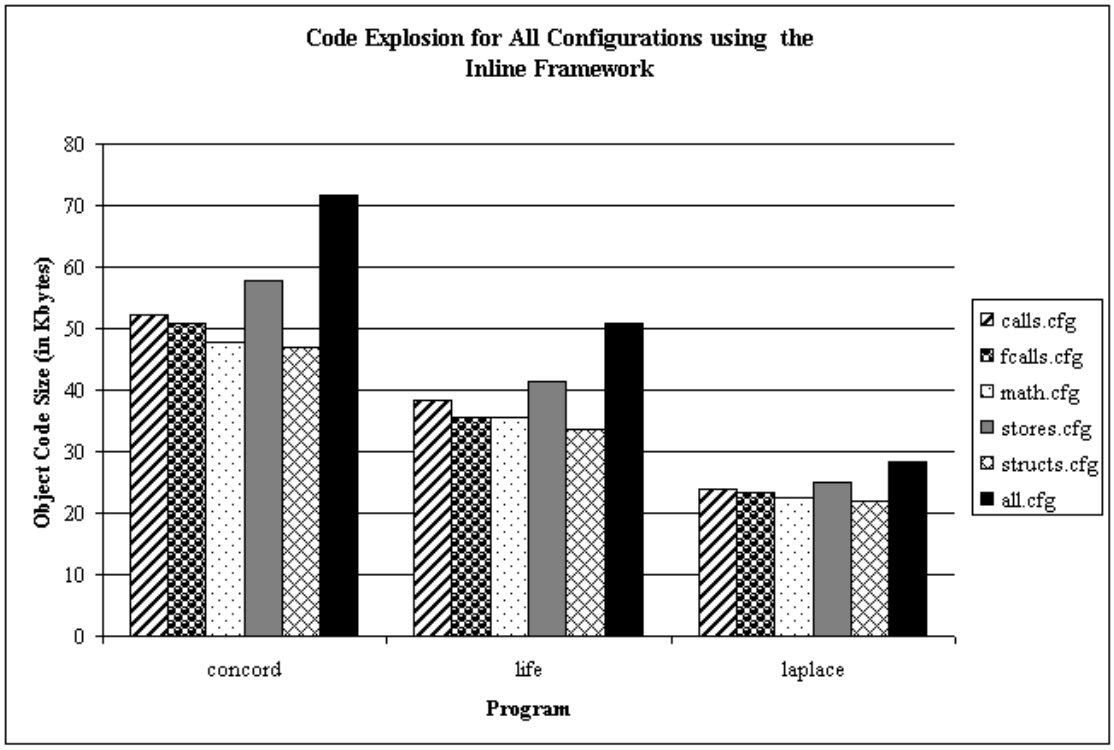


Figure 4.2: Code Explosion Measurements for the Inline Framework

Logfile framework, `concord`'s object code size increased from approximately 10K to 270K; `life`'s code explosion was from 9K to 130K and `Laplace`'s was from 7K to 90K. When various configurations were applied the average code blow up was 50K, 45K, to 40K for `concord`, `life`, and `laplace` respectively.

However, Figures 4.2 and 4.3 show little improvement in code blow up from configuring CCI with all events to only structure access events. This is not surprising because the Inline and Procedure frameworks use extremely small macros.

The code explosion shown in these results is due to the use of configurations that were for frequent, low-level events. The blow-up factors are one motiviation for refinement and for higher level instrumentation based on the results of semantic analysis, such as type information.

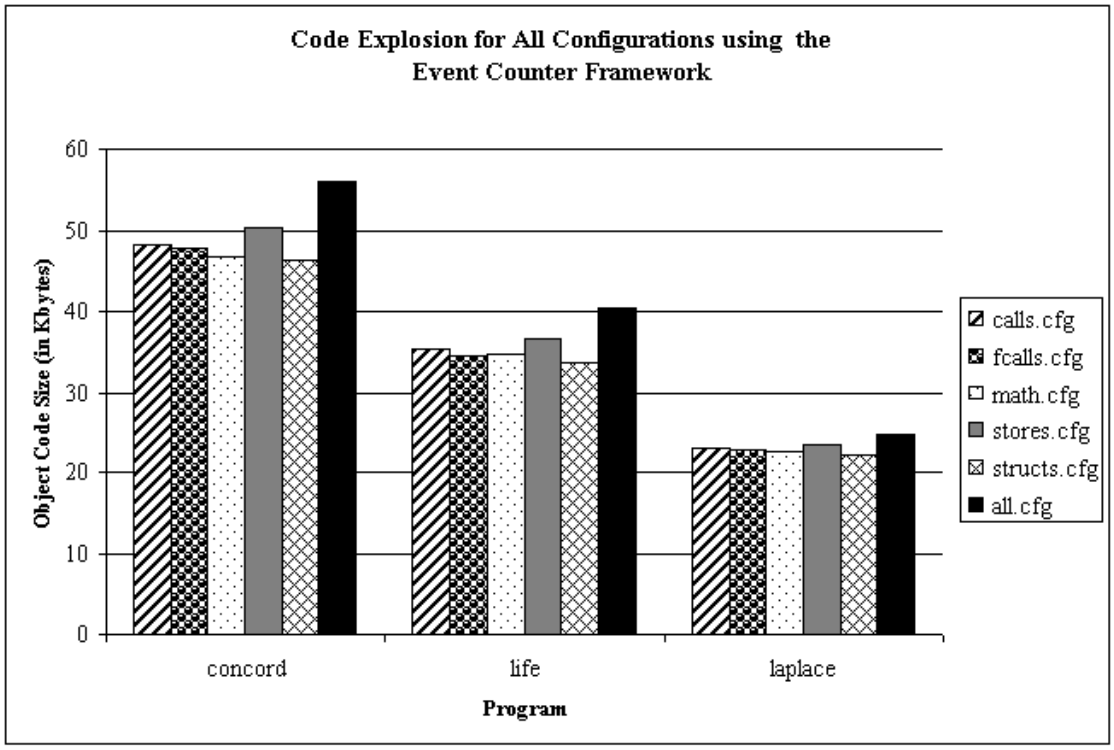


Figure 4.3: Code Explosion Measurements for the Procedure Framework

4.2 Execution Speed Measurements

In an execution monitoring environment speed is often the most critical component of usability and is more important than code bloat. Thus to make monitoring viable for any framework, the run-time cost should be proportional to the number of events required by the monitor.

To help illustrate how configuration works, Figure 4.5 shows event frequencies generated by a particular execution of the instrumented program, `concord.c`, configured to generate all events. The darkened bars shows the small fraction of function call events attributed to this particular execution. Execution time for the function call events is dominated by all the other events' execution time. This is why configuration is essen-

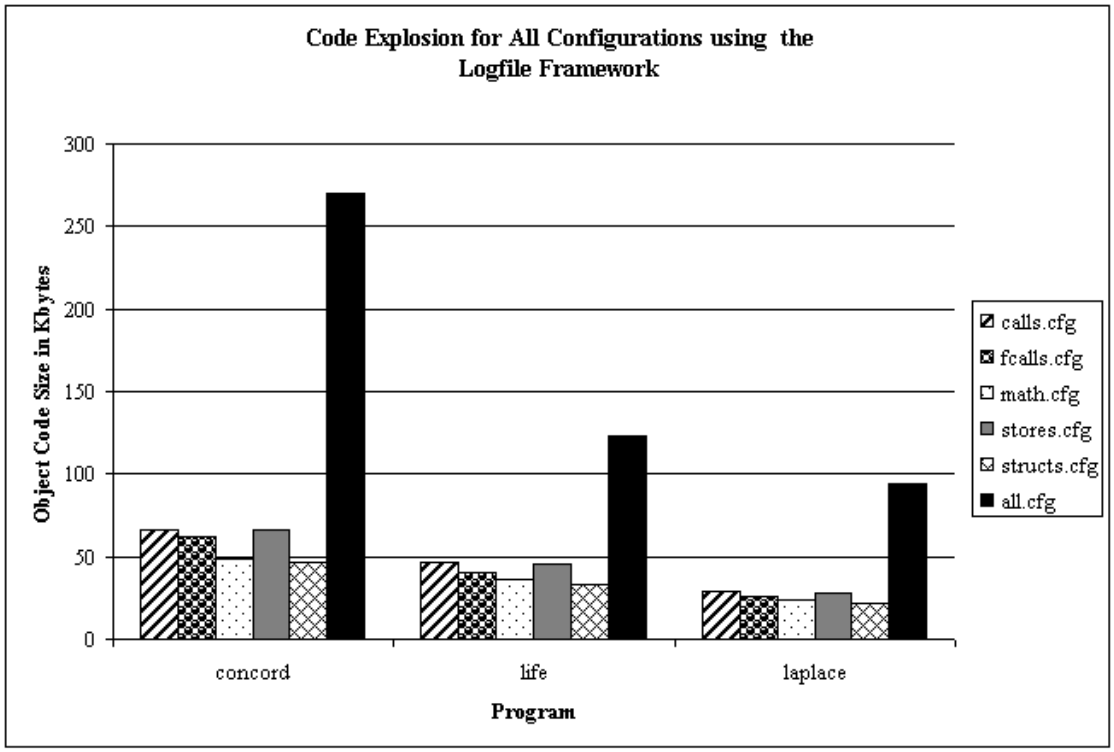


Figure 4.4: Code Explosion Measurements for the Logfile Framework

tial: By instrumenting the target program for only function calls, using calls.cfg, the only events generated will be those depicted by the black bars, considerably reducing execution-time.

Figures 4.6 through Figure 4.9 show the results of the execution time experiments. Each figure shows the execution time for a given framework across all configurations. As is expected, the bar depicting the all.cfg configuration shows that execution time is prohibitively slow because of the overwhelming number of generated events.

The effect of this is exacerbated when the macro's run-time cost is large as in the Alamo framework. For example, Figure 4.6 shows the the execution times for the programs, `concord`, `life` and `laplace` when configuring for all events. Comparing their original run-time execution, the slow down is approximately from 0.33 to 2500 seconds,

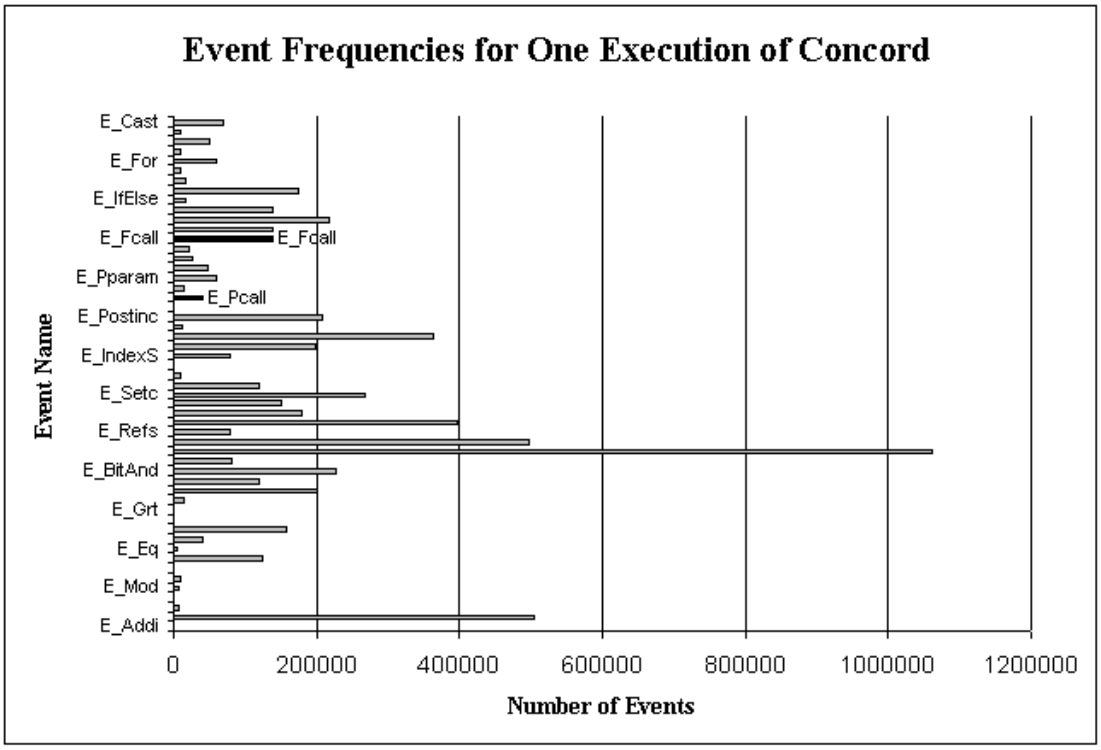


Figure 4.5: Event Frequencies for `concord.c` on CCI's `y.tab.c`

from 0.24 to 3300 seconds and 0.63 to 2600 seconds. These measurements show that in the absence of run-time filtering, instrumenting all events is not feasible. With configuration, in the worst-case example, the execution times are reduced from 2500 to 800 seconds, 3300 to 1550 seconds and 2700 to 1300. When looking at other configurations, however, CCI saves more time. For example the average run-times for all three programs not including `all.cfg` or `stores.cfg`, which represent the dominating behavior a program, yield execution times of 250, 425, and 200 seconds. These times, while still high compared to the original uninstrumented program, are due more to Alamo's context-switching than the slow down of instrumented code. Comparing the uninstrumented programs' execution times listed in Table 4.2 to those in Figure 4.7 shows that the Logfile framework for the programs `concord`, `life`, and `laplace`, when instrumented to generate every event

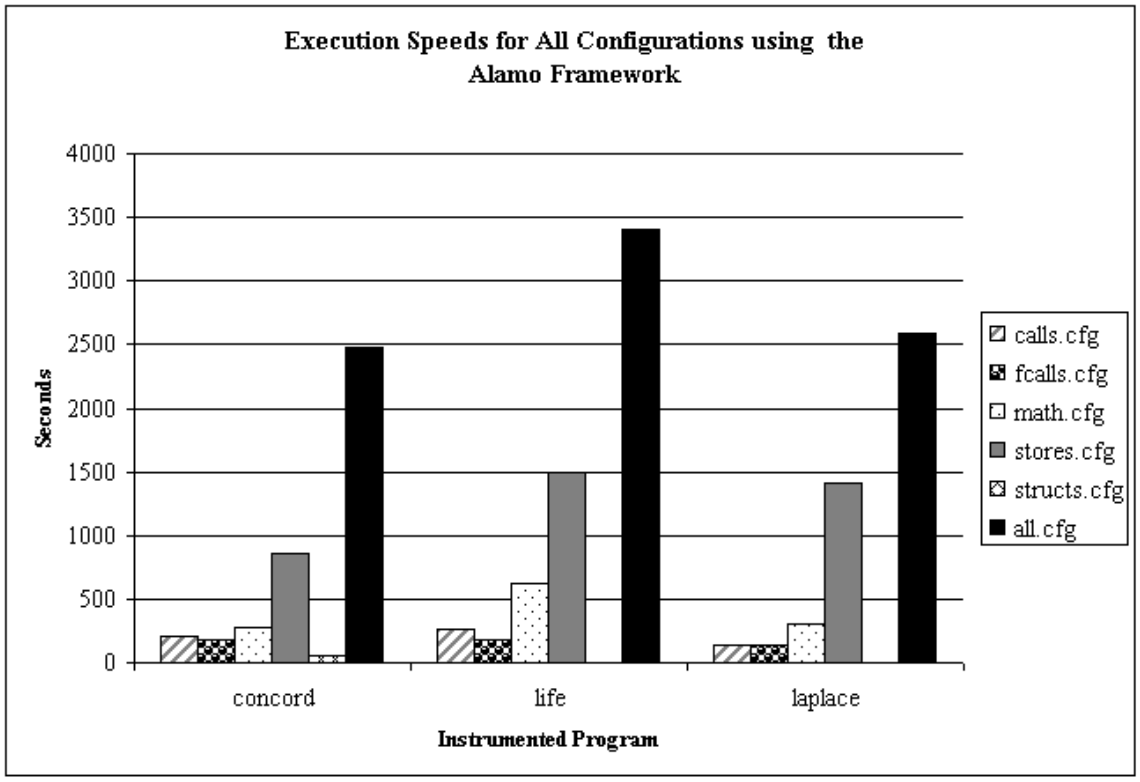


Figure 4.6: Execution Time for all programs and configurations using the Alamo Framework.

(depicted by all.cfg), yields an increase in execution times by factors of 270, 1500, 90. With configuration, the execution time increases from the original execution time, in the worst-case, by factors of 100, 330, and 115 respectively. It should be noted that these times show substantial improvement, but still are not potentially usable. However, in real monitoring environments, CCI would be configured for even smaller subsets of the configurations used in the experiments. Figure 4.8 shows execution times for the in-line framework, whose macros have an extremely low run-time cost. The execution time for each program configured for all events, is increased from approximately 0.33 to 1.2, 0.24 to 0.95 and 0.63 to 1.2 seconds. These increases are low because the run-time cost

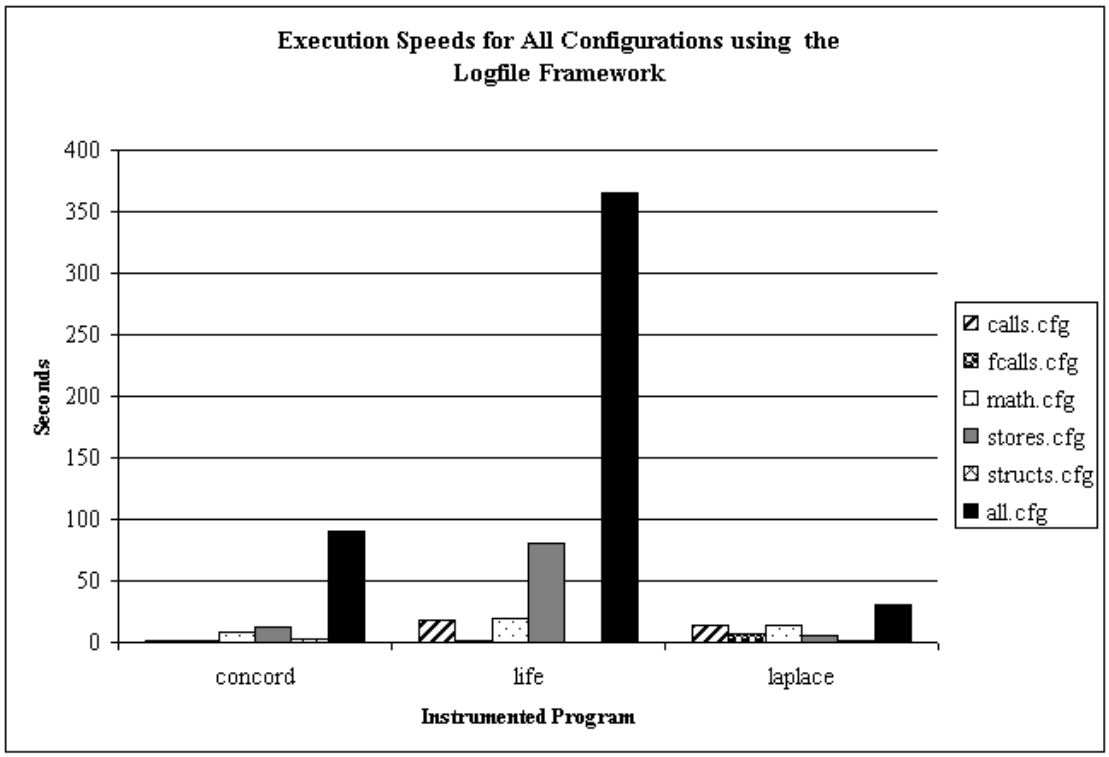


Figure 4.7: Execution Time for all programs and configurations using the Logfile.

of the macro is almost negligible and the real performance penalty here is dominated by the cost of instrumentation introduced by the pervasive assignments to temporary variables. Figure 4.9 shows execution times for the Procedure framework. Recall that this framework inserts a function call for each generated event. The execution times, when the programs are instrumented for all events, increase from 0.33 to 2.0, 0.24 to 2.2, and 0.63 to 2.45 seconds respectively. Even though these macros have low run-time cost, their total number of generated events is still very high for the all.cfg configuration. Even under these conditions, configuration still contributes to a reduction in execution time. For example, in Figure 4.9 the best case instrumented execution-time improvement compard to the original execution times is from 0.33 to 0.85, 0.24 to 0.6, and .63 to 0.90 seconds. When configuration is applied to programs that use a framework that has a

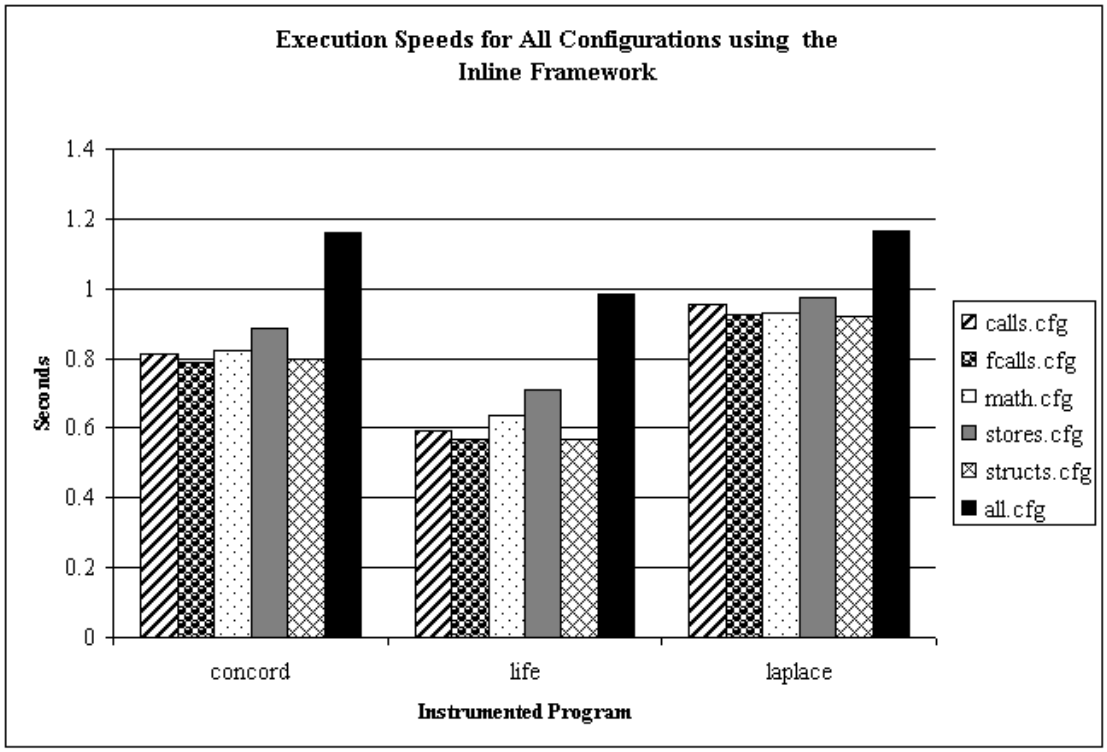


Figure 4.8: Execution Time for all programs and configurations using the Inline Framework.

low framework overhead, as in the above example, the dominating execution-time cost is that of the inserted instrumentation code. These are areas in which CCI has not been optimized and is the subject of future work.

4.3 Static Filtering Versus Run-time Filtering

As described earlier, the Alamo framework uses a run-time filter called an event mask. The Alamo macros check the current event at run-time against the event mask which is represented as a bit vector. If the event is in the mask, it is reported by performing a light-weight context switch to the monitor. If not in the mask, the event is ignored.

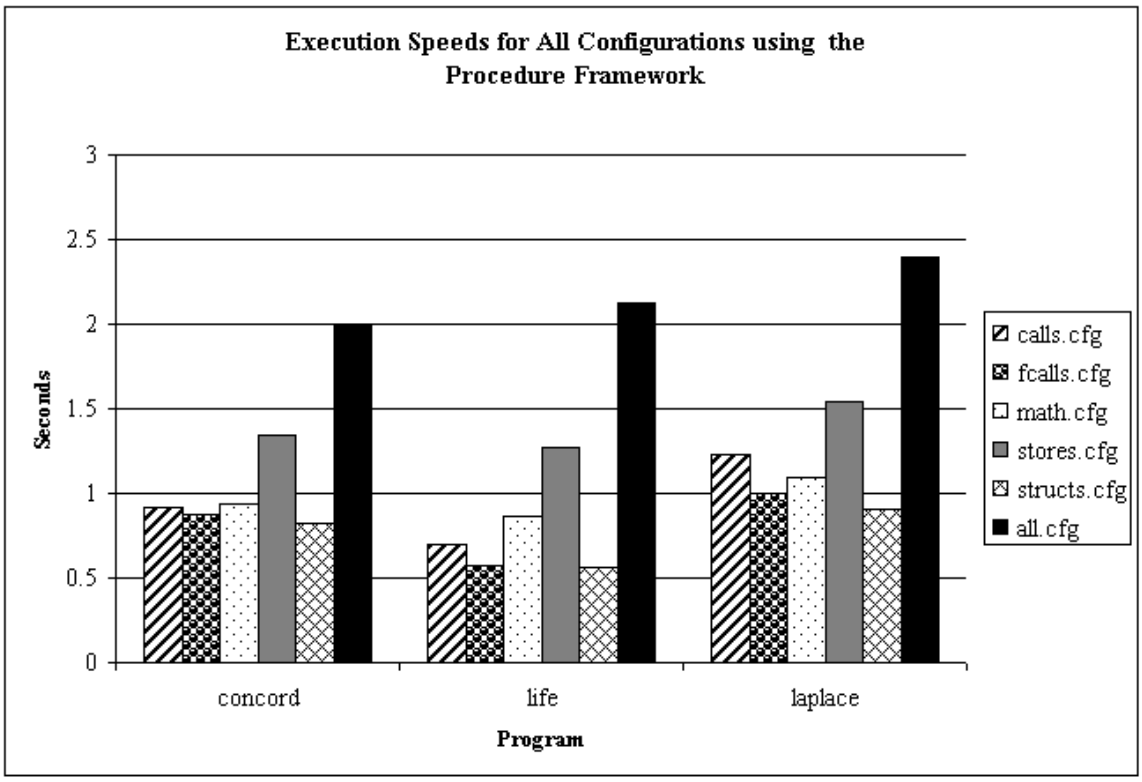


Figure 4.9: Execution Time for all programs and configurations using the Procedure Counter Framework.

In Alamo, the monitor sets up the event mask indicating the kinds of events that are needed. The monitor may at any time change the mask according to its particular needs.

Run-time filtering discards unwanted events in the instrumented code instead of performing a context-switch, saving substantial execution time. For example, if CCI were configured to generate every event and the monitor set an event mask to collect the events defined by calls.cfg, the framework's slow-down would be substantially less than if it did not perform run-time filtering. However, the Alamo framework is still performing a mask check for *every* event instrumented by CCI. This is Alamo's framework overhead cost. Using configurability, CCI can reduce the number of run-time filter checks. This section demonstrates how much savings is afforded by using configuration instead of the run-time

masks. In reality the two filtering techniques can be used together, because the user may configure CCI for the events that are of interest to any of several monitors. Then as the monitor runs, it typically needs only a subset of these events during a particular window of execution and sets the run-time filter to *further* narrow the event stream.

4.3.1 Protocol

To measure the effects of configuration over run-time filtering, CCI was configured to instrument the three target programs, `life`, `concord`, `laplace`, for all events. Five Alamo monitors were used which only maintain a total count of generated events. Each monitor was set up with an event filter mask that performs the same filtering, only at run-time, that CCI would perform if it were configured accordingly. For this presentation, the monitors are called `structs.mon`, `calls.mon`, `fcalls.mon`, `stores.mon` and `math.mon`.

4.3.2 Results

Code Blow-up

When the target programs were compiled their code blow-up for `concord`, `life`, and `laplace`, was from 20K to 600K, 17K to 420K, and 13K to 190K respectively. This increase in code size is vastly larger than those reported in the previous section. For example, when CCI is configured to instrument for `math.cfg` the code blow-up was from 20K to 130K. These results show that using run-time filtering techniques with instrumentation provided by CCI is not feasible, and that configuring CCI is the only way to make monitoring feasible for large programs.

Execution Time Performance

The monitors were run on each of the target programs five times and their trimmed mean execution time recorded. The same machine was used for these runs that was used in the previous section.

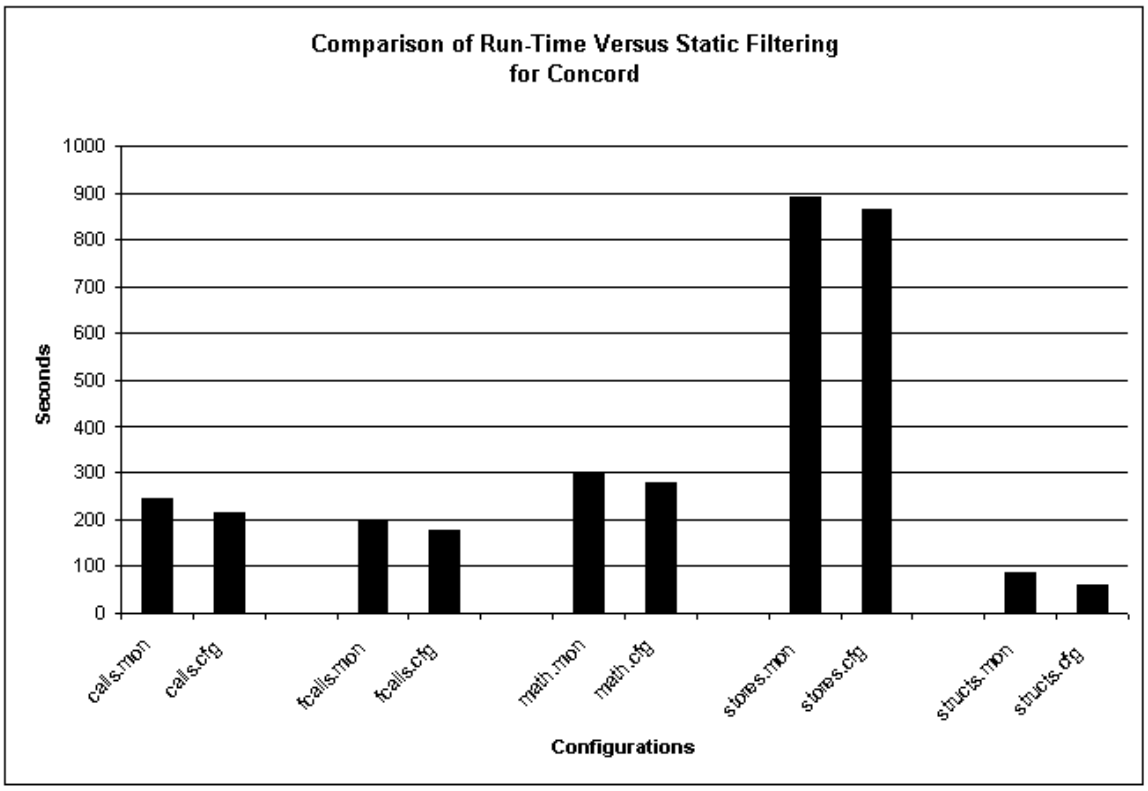


Figure 4.10: Comparison of execution time using run-time filtering or static filtering for `concord`.

Figures 4.10 is organized to compare, for `concord`, the run-time filter (indicated by the .mon extension) with the corresponding CCI configuration file. Figure 4.10 shows that marginal improvements when using configuration. The reason for the marginal improvement is that the run-time mask checks are efficient and that the context-switch time dominates the run-time execution of these programs. Thus in order for configuration

to help reduce run-time, CCI must be optimized by removing the excessive temporary variable assignments found in the current implementation.

4.4 Summary

In CCI, configurability reduces an otherwise prohibitive execution time to a usable execution time. Without configuration, frameworks which use run-time filtering would have considerabe code blow up, but with static filtering code blow-up is small enough that even large programs can be monitored. In comparing execution speeds of run-time filtering and static filtering, small improvements were realized. However, in order for there to be more improvement, CCI would have to be optimized. Another result is that frameworks which do not require special process control can insert more monitoring code and increase the monitor run-time cost without suffering as significant a performance penalty as incurred without configuration.

Although these performance measurements show the value of configuration in a broad sense, they do not convey the full potential of CCI to reduce the cost of monitoring. The use of semantic analysis in configuration features such as value masking can produce dramatic improvements for monitors that observe specific aspects of program behavior, such as heap management.

Chapter 5

Conclusions

CCI is a valuable tool for use in exploratory execution monitor development. CCI's major contributions are its configurability and flexibility. Because CCI provides a low level basis set of events, it provides fine-grained instrumentation suitable for a broad range of monitors. More importantly, the monitor writer can configure CCI to select very specific events and to generate higher level events. Of key importance is the ability to generate events based on the type of the event's value. In addition, CCI provides value masks which provide features for refining an event or renaming many different events into one event. Value masks are enhanced by the ability to create new events when an event's value matches the mask. Also, CCI's static filtering offers substantial improvements in code size when compared to run-time filtering. Moreover, because CCI allows the monitor writer both to define their own event macros and to configure CCI, it may prove to be a useful tool for many different monitoring paradigms.

5.1 Limitations

The most significant limitation of CCI is that it can not be used in frameworks which require real-time or close to real-time performance. However, CCI is useful for most other programming paradigms. A related limitation is that CCI's code blow-up may preclude its use on very large programs and the instrumented program's execution speed may not be usable for long-running programs.

CCI is piece of research software, and its other limitations are mostly implementation details. For example, CCI will fail on programs that use old-style, pre-ANSI C declarations. The declaration:

```
void main(arg, argv)
   int arg;
   char **argv;
{
   ...
}
```

causes CCI to fail. CCI also does not handle enumeration types or bit fields. No technical reason exists for these limitations. These are "loose-ends" that remain to be implemented before CCI can be considered a production-quality tool.

5.2 Future Work

5.2.1 Reducing Code Explosion and Increasing Instrumented Program Speed

Part of the future work left for CCI is to get it to work on large programs. In order for this to be possible more work must be done to minimize code explosion.

Code explosion is caused by CCI blindly inserting instrumentation for every event, even if the configuration does not select it. This instrumentation is of the form of inserting temporary variables to hold intermediate expressions. Recall that CCI implements

configuration by only altering which event macro is called. A better method is to only add instrumentation for precisely those expressions that need it in order to generate events selected via configuration. This will reduce code explosion considerably. Reducing code explosion will have the added benefit of reducing the execution time of the instrumented program. By optimizing the instrumentation, large performance gains can be made.

5.2.2 Improving Configurability

Another improvement to CCI is the ability to insert instrumentation into different scopes based on line number or function. This would allow instrumentation of only a particular function, reducing code explosion and increasing the instrumented program's speed further.

Another area of future work is to add type masks to CCI. A type mask works similarly to a value mask but instead of checking the event's value against a mask, the event value's type is checked against a mask. If the type of the value is in the mask the event is generated. Below is a syntactic example of how type masks might apply:

```
E_All{struct foo = E_FooEvent, struct foo *=E_FooEvent};
```

This would set CCI to generate events only when the event value's type is `struct foo` or `struct foo *`. When the events are generated, the event name is `E_FooEvent`. This feature would increase the selectivity of configurability for CCI, resulting in speed increases for monitors that work with such specific data types.

Appendix A

This appendix lists the set of basis events introduced in Chapter 2. Each table lists a subset of the basis events categorized by the general behavior they reflect. If an entry in a table is bold then its event value is the address of the identifier it references.

Table 5.1: Event Names for Mathematical Operators.

Event Names	Binary Operator
E_Add(i,li,f,d,ld,c,p)	+
E_Div(i,li,f,d,ld,c)	\
E_Mod	%
E_Mul(i,li,f,d,ld,c)	*
E_Sub(i,li,f,d,ld,c,p)	-
E_Neg	-
(See Table 5.2 for mathematical assignment operators	e.g. *=

Table 5.2: Event Names for Assignments

Event Names	Binary Operator or Meaning
E_Set(i,li,f,d,ld,c,p,s)	Indicates a variable is about to be set
E_InitSet(i,li,f,d,ld,c,p,s)	Same as above except these events are generated for declarations
E_AddAssign(i,li,f,d,ld,c,p)	`+=`
E_Assign(i,li,f,d,ld,c,p,s)	`=`
E_BitAndAssign	`&=`
E_BitOrAssign	`|=`
E_BitXorAssign	`^=`
E_DivAssign(i,li,f,d,ld,c)	`\=`
E_InitAsn(i,li,f,d,ld,c,p,s)	`=` (in declarations)
E_ModAssign	`%=`
E_MulAssign(i,li,f,d,ld,c,p)	`*=`
E_ShiftlAssign	`<<=`
E_ShiftrAssign	`>>=`
E_SubAssign(i,li,f,d,ld,c,p)	`-=`
E_Lvalue(i,f,d,ld,c,p,s)	any
E_Set(i,li,f,d,ld,c,p)	

Table 5.3: Event Names for Bitwise Operators

Event Names	Binary Operator
E_BitAndAssign	&=
E_BitOrAssign	|=
E_BitXorAssign	\^=
E_ShiftlAssign	<<=
E_ShiftrAssign	>>=
E_BitOr	|
E_BitAnd	&
E_BitXor	^
E_BitNot	~
E_Shiftl	<<
E_Shiftr	>>

Table 5.4: Event Names for Logical Operators

Event Names	Binary Operator
E_Or	|
E_And	&&
E_Eq	==
E_Neq	!=
E_Less	<
E_Lesseq	<=
E_Grt	>
E_Grteq	>=
E_Bang	!

Table 5.5: Event Names for Structures and Arrays.

Event Names	Meaning
E_Refs	Structure is being referenced
E_IndexS	A structure member is being referenced
E_ContC	Contents of a pointer to a struct
E_Refa	Array is being referenced
E_Index	Array is being indexed

Table 5.6: Event Names for Unary Operators.

Event Names	Meaning
E_Neg	Negation (-)
E_Addrof	`&`
E_Cont	`*`
E_Sizeof	`sizeof()`
E_Postinc	`++`
E_Postdec	`--`
E_Preinc	`++`
E_Predec	`--`
E_Cast	passes a temporary variable with the type to be casted to

Table 5.7: Event Names for Control Flow

Event Names	Meaning
E_Pcall	Application call
E_Pret	Return value from application call
E_Pparam	Parameter value for application call
E_Pstart	Start of application function
E_Pend	End of application (no explicit return expression)
E_PendExp	End of application with explicit return expression
E_Fcall	Library function call
E_Fret	Return value from library function call
E_Fparam	Function parameter event

Table 5.8: Event Names for Control Flow Continued

E_If	value of if test that has no else
E_IfExp	if expression event of the form *(exp) ? exp1 : exp2*
E_IfElse	value of if test that has an else
E_While	value of while's expression
E_WhileBegin	event before while loop starts
E_WhileEnd	event when while loop finishes
E_Do	event when do loop is evaluated
E_DoBegin	beginning of do loop
E_DoEnd	end of do loop
E_For	for expression evaluated
E_ForInit	for initialization evaluation
E_ForIter	for iteration evaluation
E_ForEnd	end of for loop
E_Switch	switch expression evaluated
E_Case	case statement (passes the label's string)
E_Default	default statement
E_Continue	
E_Goto	passes the label's string
E_Break	
E_Label	generated at the point of a label

Appendix B

This appendix lists CCI's configuration grammar. The grammar contains three unspecified terminals: *EventName*, *Identifier*, *SimpleType*. These terminals should be defined in the lexical analyzer. The terminal *EventName* is returned when the sequence of letters match one of the names in the basis set or a set name that has been previously defined or is the name of a previously defined mask. The terminal *Identifier* is a sequence of characters with similar naming rules to C. The terminal *SimpleType* is currently defined as one of the simple C types: int, char, float, long, double, etc. However, the sections of the grammar concerning type masking are in development stages and have not been implemented.

Config:

EventBlock

EventBlock FunctionSpecs

FunctionSpecs:

FunctionBlock

FunctionSpecs FunctionBlock

FunctionBlock:

`[` *Identifier* `]` *EventBlock*

EventBlock:

EventBlock EventStatement ;

EventStatement ;

EventStatement:

EventListSpec

EventSet

EventListSpec:

EventSpecifier

EventListSpec , *EventSpecifier*

EventSpecifier:

EventName

EventName MaskSet

EventSet:

Identifier =

EventListSet

EventListSet:

(*EventListSpec*)

MaskSet:

{ *MaskList* }

MaskList:

Mask

MaskList , *Mask*

Mask:

GroupMask = *MaskName*

MaskType:

Identifier

TypeSpecifier

MaskName:

Identifier

EventName

GroupMask:

MaskType

GroupMask | *MaskType*

TypeSpecifier:

SimpleType

`struct` *Identifier*

`typedef` *Identifier*

Bibliography

[Gola93] M. Golan and D. R. Hanson, DUEL — A Very High-Level Debugging Language, *Proceedings of the Winter USENIX Technical conferences*, 107-117, San Diego, CA, Jan. 1993.

[Lint90] M. A. Linton, The Evolution of Dbx, *USENIX Summer Conference*, 211-220, June 11-15. 1990.

[Holl90] J. K. Hollingsworth, B. P. Miller, J. Cargille, Dynamic Program Instrumentation for Scalable Performance Tools *Scalable High Peformance Computer Conference*, May 1994.

[Henr90] R. R. Henry, K. Whaley and B. Forstall, The University of Washington Illustrating Compiler, *Proc. ACM SIGPLAN'90*, 223-233, June 1990.

[Gris90] R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, 3rd ed. Prentice Hall, 1990.

[Braz97] M. C. Brazell, C. L. Jeffery, Tree Structure Detection and Visualization *Technical Report* **97-7**, Division of Computer Science, University of Texas at San Antonio, 1997. http://www.cs.utsa.edu/research/alamo/tr97_7

[Jeff94] C. L. Jeffery and R. E. Griswold, A Framework for Execution Monitoring in Icon, *Software: Practice and Experience*, Vol. **24**(11), 1025-1049, Nov. 1994.

[Jeff96] C. L. Jeffery, W. Zhou, and K. S. Templer, The Alamo Monitor Framework, *Technical Report*, **96-7**, Division of Computer Science, University of Texas at San Antonio, 1996.

[Olss90] 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.

[Jeff93] C. L. Jeffery, Monitoring and Visualizing Program Execution: an Exploratory Approach, Ph.D Dissertation, 1993.

[Sosi95] R. Sosic, The Dynascope Directing Server: Design and Implementation, USENIX Association, *Computing Systems*, Vol. **8**(2), 107-133, 1995.

[Sriv94] 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.

[Zhou96] W. Zhou, Implementation of the Alamo Monitor Executive, University of Texas at San Antonio, Division of Computer Science, Master's Thesis, December 1996.

Vita

Kevin Sebastian Templer was born September 13, 1969 in Lanstuhl Germany. He received his B.S. degree in Computer Science from the University of Texas at San Antonio. His interests are in compiler writing, artificial intelligence and graphics. In addition, he loves playing classical and rag-time piano.

Permanent address: 7520 Potranco Rd. #604
San Antonio, TX 78251

This thesis was typeset by the author in LaTeX.