IMPLEMENTATION OF THE ALAMO MONITOR EXECUTIVE

by

WENYI ZHOU, M.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
November 1996

Acknowledgements

I first wish to thank the members of my committee, Dr. Clinton Jeffery, Dr. Kay Robbins, and Dr. Samir Das for their knowledge and encouragement. In particular, I am very grateful to Dr. Clinton Jeffery, my advisor, supporter, critic, and friend. I am very fortunate for having the opportunity to work with him.

I would also like to thank my colleague, Kevin Templer who cracks me up all the time and kept me sane during the development of my thesis. I sure am going to miss you a lot.

My family also deserves many thanks for being supportive and made everything possible. Finally, I would like to acknowledge the members of the High Performance Computing and Software Lab at UTSA. They were always there when I needed help.

The research was supported by the National Science Foundation under grant CCR-9409082.

WENYI ZHOU

The University of Texas at San Antonio

November 30, 1996

Abstract

The Alamo system is a framework for monitoring the execution of ANSI C programs. Its design extends earlier work developed for an interpretive language by developing techniques suitable for a compiled systems programming language. The Alamo framework supports a range of dynamic analysis tools. It consists of three major components: a Configurable C Instrumentation (CCI) tool, the Alamo Monitor Executive (AME), and run-time support libraries for graphics, visualization, and target program (TP) access.

This thesis describes the AME component of the Alamo framework. The major contribution of this work is the successful implementation of the coroutine-based execution model, the context switch between the TP and execution monitors (EMs), the dynamic code loading facility, and the TP access library routines.

The implementation of the AME provides an efficient execution environment in which users can develop multiple, specialized user-level monitors that simultaneously observe a C program's execution. Monitors are dynamically loaded into the same address space as the program under observation, but retain their own locus of control and are protected from errant target program memory writes. AME also provides monitor writers with direct source-level access to the program's state and variable information at run-time. The successful implementation of the framework demonstrates that Alamo's design is feasible for applications that monitor the execution of ANSI C programs.

Contents

Chapter 1

Introduction

In the software production life cycle, programmers often encounter a need to understand the behavior of a program. This need arises in the debugging, testing, performance tuning, or maintenance of a large program. Programmers use *execution monitoring* tools to help them gain insight about a program when they cannot obtain sufficient understanding by studying the program text. Execution monitoring tools are developed to collect information from a program's execution and present that information to the user in an understandable way. Unfortunately, the field of program execution monitoring has not kept up with the rapid progress of programming languages [Plat81]. Most programming languages are not developed with monitoring capabilities in mind in the first place, and facilities generally are integrated into the programming environment as an afterthought instead.

In this thesis, a framework for monitoring the execution of programs written in the C programming language is presented. This framework is aimed at providing an environment for developing execution monitors. This chapter begins with an introduction to different techniques used for understanding the behavior of a program, followed by the discussion about difficulties encountered in the writing of execution monitors. The motivation and features of the proposed framework are discussed next. This chapter also includes a summary of the recent related work in this area. The organization of the thesis is presented at the end of the chapter.

1.1 Techniques used for understanding program behavior

The execution of a program can be thought of as moving a point along a certain trajectory through an n dimensional space. A program state is associated with the point at any instant in time [Plat81]. The trajectory is hard for people to follow and often behaves differently from what was expected. Persons who need to understand the behavior of a program usually have several alternatives:

- read the source code
- run it and view the output
- analyze computational complexity of the program
- resort to formal proofs of program correctness
- use software instrumentation techniques
- use debugger/profiler
- use visualization tools

Reading the source code is the most obvious way but is impractical for large programs. Constructing different test cases on which to run the program is also a tedious task. In most applications, enumerating all the possible combinations of inputs is impossible. Analyzing the computational complexity of a program reveals the time or space usage of the program as a function of its input, but it cannot give users detailed program state information.

A great effort is needed to prove the correctness of a program. Despite the attractiveness of the program derivation and verification approach where program correctness is "guaranteed" by formal methods, formal specification techniques are still very limited and impractical for most programs. Even with perfect derivation and verification tools, mistakes due to human errors are still inevitable.

The approach of using software instrumentation, such as inserting printf function calls in the source code, can help find errors. However output from the instrumentation can overwhelm the user and make problems hard to locate. Moreover, reducing the volume of information generated by instrumentation requires the instrumentor to have a good understanding of the program in the first place.

Using a debugger or profiler can reveal the dynamic aspects of the program behavior. Debuggers can obtain their information about the program from reading a core file after the program's execution completes, but they can also obtain the program's execution information as the program is running. On the other hand, profilers usually obtain their information from reading the log file generated by the program. They belong to post-mortem analysis tools. Using a debugger, the user needs to have a good idea about where to look for the problem. Otherwise, it may be like searching for a needle in a hay stack. Profilers are good at displaying a summary of the statistical behavior of the program, but do not provide enough detail for many program understanding tasks.

An ideal tool should be able to visually display the behavior of the program as it is executing. Users can steer the execution of the program, ask questions about the program, or even change the values of some variables in the program without imposing too much of a delay on the program being monitored. Such a visualization tool must collect extensive information about a program's execution.

1.2 Problems faced when writing visualization tools

Writing efficient visualization tools for monitoring the execution of a program is a complex task. The major problems encountered in obtaining execution information are *volume*, *intrusion* [Henr90], and *access*. [Jeff93] has detailed descriptions of those problems. Here we only emphasize that efficient gathering of information and processing large amounts of data without obscuring items of interest is the basis for a good execution monitoring tool. Also a good execution monitoring tool should be able to minimize the intrusion effect to the program being monitored both code-, data-, and speed-wise. Finally a good execution monitoring tool should be able to navigate the execution of the program being monitored and be able to collect variables and state information from it easily.

1.3 Motivation of the framework

Due to the universal problems mentioned in the previous section, development of execution monitors is very difficult. Much effort must be expended solving those problems whenever a monitoring tool is written. Alamo is a framework that supports a range of dynamic analysis tools. It runs on Sun workstations that use the Solaris 2.x operating system. The goal of the Alamo framework is to reduce the difficulties in writing monitoring tools by constructing a platform on which monitor construction is relatively easier. (Alamo stands for A Lightweight Architecture for MOnitoring.) Alamo's design extends earlier work developed for an interpretive language [Jeff93] by developing techniques suitable for a compiled systems programming language, ANSI C.

Applications of this framework can be in a variety of areas. Execution monitoring tools developed using this framework can help programmers in debugging their programs, performance tuning their programs, visualizing their programs, or simply, trying to understand program behavior. Given this framework, the development of such tools is no more difficult than writing application programs that involve communication between programs.

1.4 Features of the Alamo framework

The following terms are used throughout this thesis for describing the framework.

target program (TP) — a C program that is being monitored. It is a relocatable object file.

execution monitor (EM) — a C program that collects and presents information from the execution of a TP.

alamo monitor executive (AME) — a C program that dynamically loads the TP and one or more EMs.

The Alamo system consists of an execution model, an automatic instrumentation mechanism with event configuration ability, and run-time support libraries for graphics, visualization, and TP access. This thesis describes the implementation of Alamo's execution model and TP access library. The instrumentation mechanism and configurability of events are described elsewhere in [Temp96]. The most important feature of the framework is that it applies the microkernel approach to solve the execution monitoring problem. Instead of writing monolithic monitoring systems, it encourages users to develop specialized tools that observe specific aspects of program behavior. The following sections describe the features associated with this design choice in more detail.

1.4.1 Multi-tasking

The first important decision in the design of Alamo is to choose an execution model from the three primary execution models used by most monitoring tools. Those primary models are the *one-process* model [Brow84], *two-process* model [Sosi1], and *thread* model [Aral88]. Detailed descriptions of these models can be found in [Jeff93]. The Alamo framework uses the thread model. The reason for this is that the one-process model is too code and data intrusive and the two-process model is too slow because of the switching back and forth between TP and EM. The thread model lies midway in between these two models. In the thread model, the TP and EM are two different entities but they share the same address space. Context switch time between them is much less than that between two different processes and accessing the TP's information is much easier for EMs.

Because threads execute concurrently, a standard thread model would allow the execution of TP and EMs to be truly parallel on multi-processor machines. If the TP and EMs execute simultaneously, the TP might modify state while it is being accessed by EMs. In Alamo the TP and EMs are synchronized in order to ensure the consistency between them. This has the advantage of giving EMs full stable information about the behavior of the TP and equally importantly, it substantially simplifies monitor development. The precise nature of the interaction between the TP and EM is discussed further in Chapter 3.

1.4.2 Dynamic loading

Since multiple tools can be used to monitor the same program, users need the flexibility of choosing what tools to use and how many of them to use. This framework provides users with this kind of flexibility by implementing a dynamic loading facility. *Dynamic loading* is the ability to load multiple programs into a shared execution environment. It is important to note that dynamic linking is not desirable in the context of execution monitoring, since the names in EMs are distinct from those in the TP. Because of the dynamic loading feature of the framework, users can pick EMs at run-time without recompiling the program.

The AME performs the task of loading multiple EMs and the TP. All of the loaded programs occupy separate memory regions and they can also allocate memory from separate heaps. For this reason, memory allocations in the EM will not affect memory allocations in the TP. In addition, memory pages that belong to the EM can also be protected from those of the TP in order to prevent program corruption. Chapter 2 will discuss the implementation of this dynamic loading facility in more detail.

1.4.3 Information sources and access methods

Many monitoring tools would be incomplete without the ability to fully access the TP's state information. Several methods are used to obtain information about the program behavior during execution. The quantity and quality of the monitoring that can be performed are determined by how much information can be obtained and how hard it is to obtain. The most common ways for gathering information are through *manual instrumentation* [Brow84], *run-time instrumentation* [Laru95], and *language-supported instrumentation* [Henr90]. Language-supported instrumentation includes *preprocessor instrumentation* and *interpreter instrumentation* [Jeff94].

Manual instrumentation of each program being monitored is too labor intensive. It adds additional effort when an instrumented program is modified. The information obtained from run-time instrumentation is usually too low-level. The modification is done at machine language level and is inevitably machine dependent. For these reasons, Alamo takes a language-supported instrumentation approach. Since the C language is a compiled, systems programming language, a preprocessor is implemented for translating a C program into an instrumented C program while maintaining the semantics of the original program. The instrumentation is done using a compiler-style syntactic and semantic analysis of the source program, and allows higher-level semantic information to be introduced for common run-time libraries or individual applications. More descriptions about the different aspects of those methods can be found in [Jeff93].

The source code that comes out of the preprocessor contains extra expressions that generate *events*. An event is defined as any change of state that has information associated with it at the source-language level [Temp96]. It is the smallest unit of execution behavior that is observable by a monitor. From the monitor's view an event has two components: an *event code* and an *event value*. The code describes what type of event has taken place and the value is a C value associated with the event. The nature of an event value depends on the corresponding event code [Temp96]. Since many events are very low-level, users do not need such fine grained events most of the time. A configuration file allows users to specify the set of events that are actually instrumented. Event masking is another way used to allow users to further specify what events are actually being reported by the TP and what events are not being reported. Those methods give users the control over how much information flow they want from the TP to the EM which in turn reduces the volume.

While the TP passes its information to the monitor through events, the monitor can also peek into the TP using the library functions provided by the framework. Those library functions give EM authors more power in steering the execution of the TP. It is important to note that since EM and TP exist in the same address space, access to the TP's variables and data is direct and efficient for the EM. Accessing program variables and data from outside the address space is slow and costly as it requires operating system assistance.

1.4.4 Multiple monitors and monitor coordination

One benefit from the dynamic loading capability of the framework is that multiple monitors can be used to monitor a single TP. Since EMs are easier to write if they do not need to be aware of each other, this leads to the construction of a *monitor coordinator* (MC).

The MC is a special purpose monitor that monitors a TP, and provides coordination and monitoring services to additional EMs. Execution monitors receiving services from a MC need not be aware of the existence of the MC or the other execution monitors. Figure 1.1 shows a typical layout of multiple EMs with a MC and a TP. A full description of the synchronization and coordination between them can be found in Chapter 3.

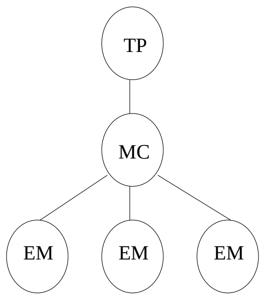


Figure 1.1: A typical configuration for multiple EMs with a MC and a TP

1.5 Related work

Earlier work in program execution monitoring is summarized in Jeffery's dissertation [Jeff93]. Since then, several new systems have been developed. In this section, a genuine dynamic linking facility (DLD) [Ho91], a program directing server (Dynascope) [Sosi1], a machine-independent debugger (cdb) [Hans96], a flexible interface for building program analysis tools (ATOM) [Sriv94], and an object-oriented application framework for the dynamic analysis of distributed programs (BEE++) [Brue93] are described.

1.5.1 DLD

The DLD [Ho91] system is able to add compiled object code or remove such code from a process any time during its execution. It performs loading of object files, searching libraries, resolving external references, and allocating storage for global and static data structures at run-time. It differs from other dynamic linkers in that not only can object modules be added to but they can also be removed from an executing process. DLD is especially useful for highly interactive programs whose functionalities change in response to their user's inputs. It is used in the source-level debugger Dalek [Olss90] as users can dynamically link debugging functions into the program being monitored. In the Alamo framework, dynamic linking is not employed since TP and EMs have separate name spaces.

1.5.2 Dynascope

The Dynascope [Sosi1] is a directing platform for monitoring and controlling C language programs. It uses the terms *director* and *executor* for the roles played by EM and TP in the Alamo framework. Director and executor are separate processes that can run on different processors, communicating using event streams across a network. Dynascope has two major components: a *client library* and a *directing server.* The client library provides a set of procedures which can be called by directors to send requests to the executor. The directing server is implemented as a distinguished thread running in the executor's address space. It isolates system dependent features and provides a system independent interface to the client library [Sosi2]. The context switch from the executor to its directing server can be caused externally or internally. The external activation is done by a director through issuing a directing request to the directing server. The internal activation is done when the executor encounters a breakpoint. At most one director can be attached to an executor at the same time, but a director may direct several executors in the distributed computation environment.

Since Dynascope director and executor are two independent processes running on separate machines, their connection has to be through stream-based interprocess communication. In addition, event streams are generated at the machine instruction level. This means that the directing service would impose significant time penalty on the program execution. In summary, this directing platform provides tracing primitives as well as access to program's variables, dynamic data structures, and its internal state.

1.5.3 BEE++

BEE++ [Brue93] is an object-oriented application platform developed at CMU for software-based distributed dynamic analysis tools. It is an event-based system in which the target program is instrumented with event sensors to denote the occurrence of specific events. Upon encountering the event sensor, an event is generated and one or more tools connected with the system is notified. The distinct feature of this system compared with ordinary event processing systems is the symmetric peer-peer architecture, in which both analysis tools and the target program can generate events and interpret events. The connection between the target program and analysis tools is through TCP/IP protocol since they can be distributed across different machines. BEE++ provides customizability of the event processing through inheritance. Users can derive customized graphical debugging and visualization systems from a set of base classes. This is a natural solution for a system implemented in C++ using OMT methodology. Its earlier software development platform BEE, which was implemented entirely in C, supports the customizability by providing a system of templates for all the tools and the target program [Brue90]. In the Alamo framework, abstractions of lower-level or higher-level events are done through a configuration file.

1.5.4 ATOM

ATOM [Sriv94] is a system that provides a simple interface to OM [Sriv93] (a link-time code modification system) for adding instrumentation to programs. ATOM tools traverse an application, find interesting places to add calls to analysis procedures, and pass arguments that correspond to data or events in the application. Instrumentation is inserted only when necessary to gather the statistics. Communication of data to the analysis procedures is accomplished through procedure calls, rather than relying on the expensive interprocess communication. As ATOM works on object modules, it is independent of compiler and language systems. ATOM's interface is higher-level than EEL's [Laru95]. This simplifies the tool writing, but provides less control over the instrumentation process. Both of these tools change executable code by removing existing instructions and adding *foreign code* that observes or modifies a program's execution. It is an effective technique for measuring program behavior since editing them does not require source code or modifications to compilers and linkers. However, instrumentation can only be inserted before or after instructions, basic blocks, and procedures. This makes obtaining higher-level information about the program difficult.

1.5.5 cdb

cdb [Hans96] is a source-level debugger for ANSI C programs compiled by lcc [Fras91], a retargetable C compiler. Most source-level debuggers are machine-dependent programs. cdb disentangles the machine-dependent parts of a debugger from its machine-independent parts. cdb achieves target independence by embedding a small amount of itself as a set of procedure calls in the program to be debugged and by having lcc emit target-independent symbol tables and breakpoint "hooks". These procedure calls are referenced as nubs. The nub serves as a communication channel between the target program and the debugger. It transfers data with its format and interpretation being agreed upon. The nub depends only on the compilation and operating systems. It has no dependencies on the target architecture. cdb can be loaded with the target in a single process on systems that do not support processes or interprocess communications well.

The default configuration of cdb is a two-process configuration. The nub is loaded with the target and communicates with cdb using remote procedure calls (RPCs).

1.6 Organization of the thesis

This thesis is organized as follows: Chapter 2 presents a novel dynamic loading facility which is used by the framework to load execution monitors at run-time without recompiling the program. Chapter 3 discusses the implementation of the synchronous coroutine execution model. Chapter 4 describes the facilities that execution monitors use to access the state information of the program being monitored. Chapter 5 gives some sample execution monitors to demonstrate the viability of the framework. Chapter 6 presents the preliminary performance results for this framework and discusses the primary issues associated with it. Chapter 7 concludes this thesis with possible directions in which the framework can be improved and its limitations.

Chapter 2

Dynamic Loading Facility

Execution monitors and target programs in the Alamo framework [Jeff96] are ELF object files [Elf93]. They are loaded into separate areas that are dynamically allocated within the address space of the Alamo Monitor Executive. The variables in loaded programs get relocated but are not linked with the other programs' modules. The only remaining external references are shared library function calls. Those loaded programs act as separate entities and do not contend with each other. This dynamic loading facility can be used not only in the monitoring of programs, but also as a general platform of running several programs simultaneously in one address space. Because those programs exist in the same address space, context switching time between them is far less than between different processes. Also passing information between those programs is easier and has lower overhead than communicating between processes.

This chapter starts with a brief summary of the ELF object file format, followed by sections that describe the AME's dynamic loading facility. It addresses the differences between our facility and the system's existing dynamic loading facilities and describes detailed implementation issues related to this unique facility.

2.1 ELF object file

The Executable and Linking Format (ELF) is a binary format originally developed and published by Unix System Laboratories [Elf93]. It is the default binary format for the executable files used by many modern variants of UNIX such as SVR4, Solaris 2.x, and Linux 1.3.x. ELF is more powerful and flexible than the a.out and COFF binary formats.

There are three main types of ELF files:

- An *executable* file contains codes and data suitable for execution. It specifies the memory layout of a process.
- A *relocatable* file contains codes and data suitable for linking with other relocatable and shared object files.
- A *shared object* file contains codes and data to be statically or dynamically linked with a process.

In ELF executable files, the addresses of variables are absolute memory locations, and cannot be relocated dynamically. An object file on the other hand can be relocated. Thus AME's dynamic loading facility requires monitors and target programs in relocatable object format rather than executable format. Collections of object files that make up an entire program in a relocatable object format are produced using the **-r** option during linking.

Figure 2.1 shows the layout of an ELF object file. An ELF object file contains an ELF header, a section header table, and several sections. An *ELF header* resides at the beginning and describes a "road map" of the file's organization. The first few bytes of the ELF header specify how to decode and interpret the file's contents. Following that, it specifies the type of an object file, the required machine architecture for an individual file, and the version number for the object file. It also gives a summary of the number of entries in the section header table, the section header's size in bytes, the file offsets for storing the section header table, *etc.* A *section header table* contains entries describing each of the file's sections. Each entry specifies information including the section name, the section size, and so forth. *Sections* hold the bulk of object file information, such as instructions, data, symbol table, string table, relocation information, and so on. Some sections occupy memory during process execution, while some control sections do not reside in memory. AME loads only those sections that are needed during program execution. There is one special section .bss which holds uninitialized data. This section occupies no file space. The section header of this section specifies the size of this section. AME leaves that much space for this section at the end of the program's image and initializes memory to zero.

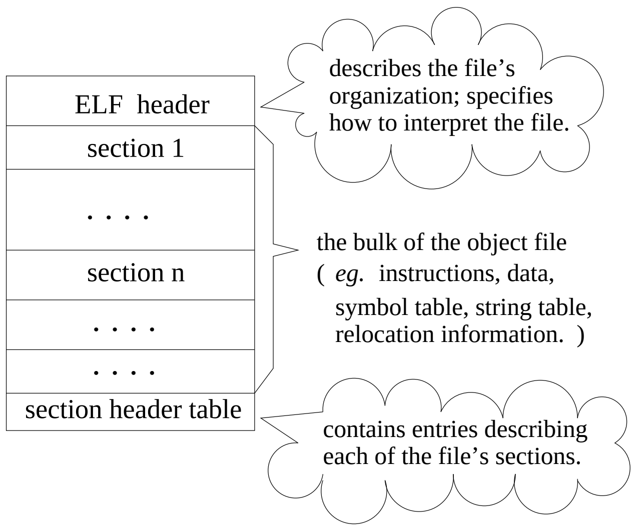

ELF header
section 1
. . . .
section n
. . . .
. . . .
section header table
describes the file's organization; specifies how to interpret the file.
the bulk of the object file (eg. instructions, data, symbol table, string table, relocation information.)
contains entries describing each of the file's sections.

Figure 2.1: The layout of an ELF object file.

Among all the sections in ELF objects, understanding the .strtab, .symtab, and .rela sections is especially important in order to load an object file correctly. The object file uses strings stored in the string table section, .strtab, to represent symbol names associated with symbol table entries. The object file's symbol table section, .symtab, holds information needed to locate and relocate a program's symbolic definitions and references. The object file's relocation section, .rela, contains information for relocating symbolic references in a program. The detailed description of their functions during loading will be discussed in later sections.

2.2 Dynamic loading without linking

A novel dynamic loading facility was developed for the AME. It differs from the system's dynamic loading facilities in that the latter one dynamically links code as it is loaded. The dynamic loading facility used by AME loads relocatable object files without linking their global variables with variables in other programs. Thus both the TP and EMs can use the global variables with the same name without interfering with each other.

There are three problems that must be addressed in order for the AME to correctly transform an object file into an executable file. They are listed as follows:

The first problem is to resolve the shared library function calls and system calls. Those function calls in the loaded programs need to be linked with the shared library in order for them to make any library function calls. One obvious way to solve this is to statically link in the shared library code at compile time. Those function calls then can be resolved by the system's static linker. However, this approach is impractical because many libraries (such as those relating to graphics and window systems) are huge. Statically linking all the shared libraries would blow up the final object file size. Furthermore, if several EMs are needed for monitoring the TP, each EM would have a copy of libraries. The resulting waste of space would be large enough to limit the scalability of the framework to large numbers of monitors. On the other hand, without statically linking non-reentrant library routines with the loaded programs, the framework has limitations in using them in the monitors. This limitation is caused by the use of static variables in those non-reentrant routines. Detailed reasons will be explained later section.

The second problem is that loaded programs (TP, MC, or EMs) need to share some important global variables and functions in the loading program (AME). Those values have to be passed to loaded programs even though there is no linking procedure between loaded programs and the loading program.

The third problem is how to manage the heap space of loaded programs. There are two alternatives. One way is to share a single system heap for all programs. The other way is to provide an independent heap space for each loaded program. The advantages and disadvantages of using these two methods are discussed later.

2.2.1 Shared libraries – procedure linkage table

Although monitors are not linked with other programs when they are loaded in, they are linked to shared libraries. When the link-editor (ld) generates the executable file image for a program, it supplies additional relocation information for shared library function calls required by the run-time linker. The procedure linkage table (PLT) contains that additional information as a part of an executable file. The PLT is used to redirect position- independent function calls to absolute locations. Its existence in the executable file assists the system's run-time linker to resolve those function calls in the program. In an ELF executable, the first four entries of a PLT are reserved for the run-time linker's own routines. All other PLT entries are constructed so that when they are called, control is passed to the run-time linker. The run-time linker looks up the required symbol and then rewrites the PLT entry using the symbol's address. Thus any future calls to this PLT entry will go directly to the function. Figure 2.2 demonstrates the procedure for the run-time linker and a program cooperating to resolve a shared library function call, *e.g.* printf.

Since we are unable to use the system's dynamic loading facilities, AME is unable to provide a new PLT for each monitor. The addresses of the library function calls in monitors have to be resolved by using the PLT entries in the AME. The .symtab section

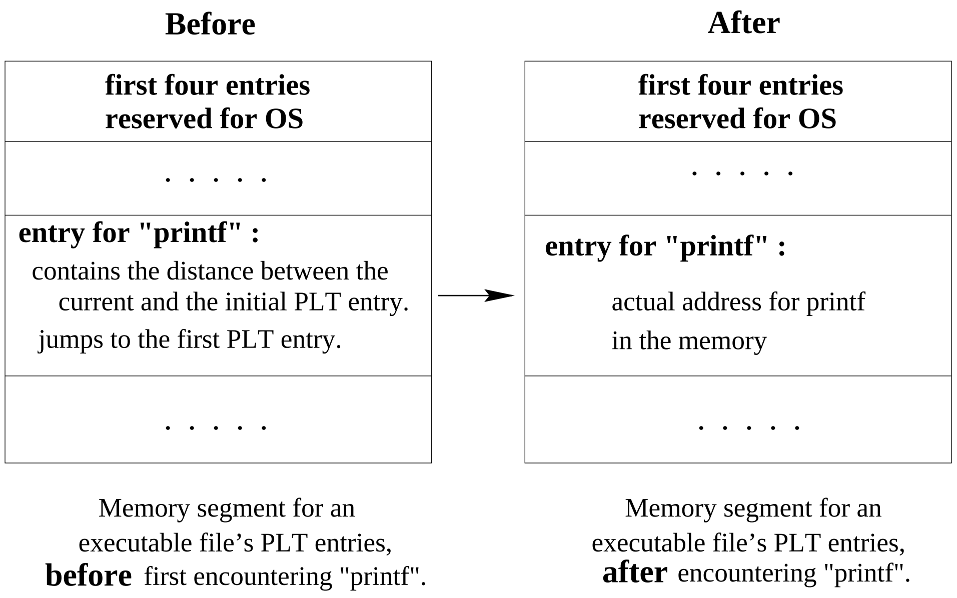

Before
After
first four entries reserved for OS
. . . . .
entry for "printf" :
contains the distance between the current and the initial PLT entry.
jumps to the first PLT entry.
. . . . .
first four entries reserved for OS
. . . . .
entry for "printf" :
actual address for printf in the memory
. . . . .
Memory segment for an executable file's PLT entries, before first encountering "printf".
Memory segment for an executable file's PLT entries, after encountering "printf".

Figure 2.2: A procedure for the run-time linker to resolve a shared library function reference, *e.g.* printf, using the PLT.

in the AME's executable file contains entries for all the global variables and function calls used. The symbol table entry is a structure containing the name, type, binding, and address information of those variables. Its format is as follows:

```
typedef struct{
        Elf32_Word st_name;
        Elf32_Addr st_value;
        Elf32_Word st_size;
        unsigned char st_info;
        unsigned char st_other;
        Elf32_Half st_shndx;
} Elf32_Sym;
```

For those variables that need to be relocated by the run-time linker, their st_value field contains the address corresponding to their PLT entry location. In order to resolve shared library function references that are used in the TP and monitors, those variables' st_value field gets the address of their corresponding PLT entry location in the AME's executable program image. Detailed explanation is presented later in Section 2.4. This way whenever a library function call is encountered for the first time in the TP or monitors, the control is transferred to the run-time linker. The run-time linker looks up the address for the symbol and transfers control to that desired location.

AME includes a dummy object file that references all the shared library calls. When it loads a program, AME resolves the loaded program's library function calls by referring to its own PLT entries. Since the system's run-time linker doesn't allow adding new PLT entries to the existing PLT, if a library function call is absent from the AME's PLT, that function call is written to that dummy file and the AME rebuilds itself.

AME has problems in using non-reentrant library routines in loaded programs, since loaded programs share the same copy of library routines used by the AME. Because shared library routines are not statically linked with each loaded program, there is only one copy of static variables used by those routines. Suppose a loaded program is in the middle of executing a non-reentrant library routine, *e.g.* strtok, and then the execution is transferred to another program which calls the same library routine. This causes that values of static variables used in strtok for the previous program are overwritten by the subsequent call. In order to avoid this problem, non-reentrant library routines are not allowed in EMs.

2.2.2 Information passing between programs

Because the loaded programs are not linked with the AME, one way for them to share important variables is to pass them through the EM's or TP's main function parameters. Those variables include the location of the heap for each loaded program, the global TP and MC program's status, and addresses of access functions [Zhou96], *etc.* The access functions are used by EM writers to obtain TP's variables and structure information while processing the events. All loaded EMs obtain the addresses of those functions by the passed data from the AME. A C structure is created to hold those variables. It is appended to the end of the original function's argument array. Figure 2.3 shows the layout of the modified argument array for the main function of a loaded program. As shown in the diagram, the additional element is appended after the NULL, so the loaded program doesn't detect that its argument array (argv) has been changed. The original argument count (argc) variable is also unchanged, thus the loaded program's behavior remains unchanged.

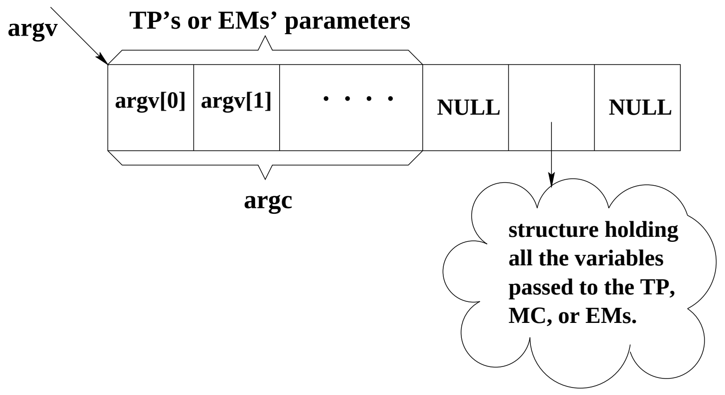

argv
TP's or EMs' parameters
argv[0]
argv[1]
. . . .
NULL
NULL
argc
structure holding all the variables passed to the TP, MC, or EMs.

Figure 2.3: The layout of the modified argument array for a loaded program's main.

Functions TpInit(argc, argv) and EvInit(argc, argv) are used by TP and EM to initialize the execution monitoring, respectively. argc and argv are the TP's or the EM's main() function parameters. These two functions are called in the very beginning of the program.

2.2.3 Heap space management

AME provides two approaches for managing the heap space of loaded programs. The first approach uses a single system heap for loaded programs. This is done by linking the malloc function used in loaded programs to PLT entries in the AME as shown in Figure 2.4. The physical memory layout of the AME is shown in Figure 2.5. AME allocates space for TP's and EMs' code, data, and stack regions. They all share the AME's heap space which is the system's heap. The advantage of this approach is that the heap space is not limited for the loaded programs, since they are sharing the system's heap space. The disadvantage of it is that the heap space allocated for the TP is interleaved with the heap space allocated for EMs. If the TP is buggy, random pointers can write over EM code or heap regions. Memory protection can be provided on EM code and stack regions, but protecting EM heap regions is impossible in this approach because of memory interleaving.

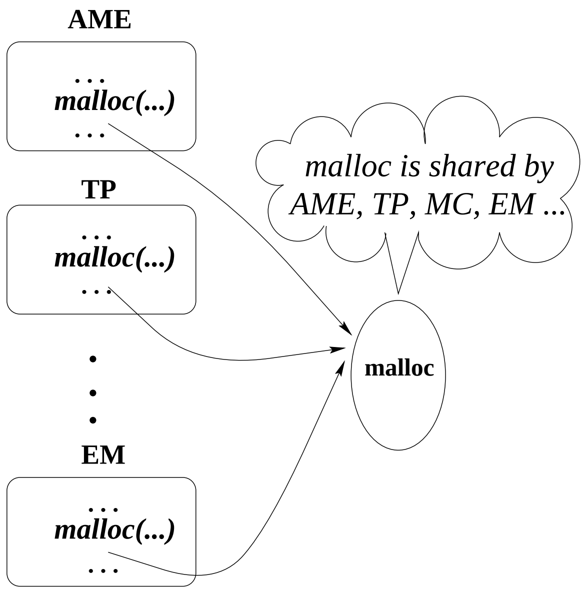


Figure 2.4: All loaded programs and the AME share a single copy of malloc function.

The second approach provides an independent heap space for each program, giving monitor writers the ability to specify memory protection on both EM code and heap regions. The detailed description of the memory protection is discussed in Section 2.3. The disadvantage of this solution is that the fixed-sized heap space represents a limitation imposed by the monitoring framework at the beginning of execution. The heap size can be set to a large number without recompilation and it is not a problem in practice.

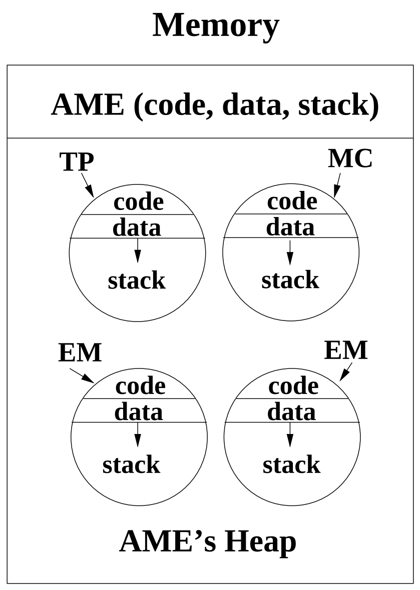


Figure 2.5: The physical memory layout for the AME using a single heap approach.

Providing independent heap space is trickier than it sounds. The loaded program has to know where its heap space is located. By studying the GNU version of the malloc function, it was discovered that malloc calls the sbrk system call if it needs to change the amount of space allocated for the calling process's data segment. GNU malloc also uses a set of global variables to keep track of the free list and used list for the memory used by the program. Since each loaded program has its own heap, they have to have their own set of these variables in order to maintain their own heap space. AME uses a modified version of the GNU malloc function. It calls the system's sbrk function if the AME needs to expand its data segment, while it calls the AME mysbrk function if the loaded programs need to expand their data segments. AME mysbrk function knows the location of each loaded program's heap. Also different sets of global variables are used for each loaded program. The modified malloc function is linked with the AME at compile time instead of the malloc function in the shared library. Figure 2.6 shows the AME, TP, and EMs share the same modified GNU malloc function. Each loaded program has its own heap space management. If the heap space is not large enough, AME displays a warning message indicating out of heap space and forces the AME to allocate more heap space. The physical memory layout of the loaded programs within the AME is shown in Figure 2.7. AME allocates space not only for code, data, and stack regions for each loaded program, it also allocates a huge heap space for them. In this way the loaded program can live within its own space and does not contend with other program's memory space.

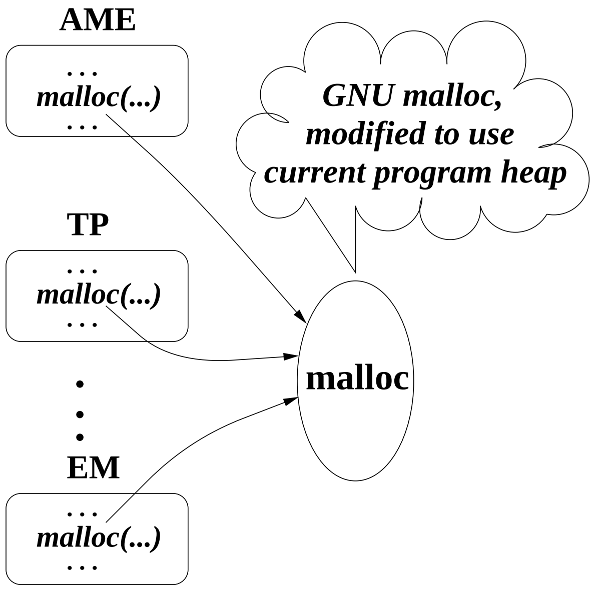


Figure 2.6: AME, TP, and EMs share the modified GNU malloc function.

2.3 Memory protection

TP and EMs coexist in the same address space. The TP being monitored could potentially be buggy. Its random pointers might write over the memory space allocated for EMs. In order to prevent TP from trashing EMs and vice-versa, memory protection between them can be implemented for the independent heap space approach. In order

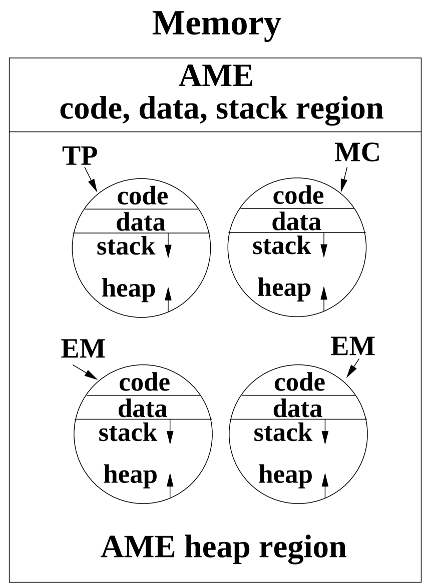


Figure 2.7: The memory layout of the loaded programs in the AME using independent heap space approach.

to do so, AME loads the TP and EMs starting at a page boundary. The system function mprotect is used to change the access protections on pages specified by its parameters. Each time after the monitor coordinator (MC) transfers control back to the TP, memory pages for the MC and EMs are mapped to be readable and executable. Similarly, before the TP gives control to the MC, memory pages of those monitors are mapped to be readable, writable, and executable. Since memories for EMs are contiguous, only one mprotect function is needed for each EM. The overall impact of this memory protection on the execution speed depends upon the frequency of context switches between the TP and EMs and the size of the memory that needs protection. Memory protection hurts performance significantly if there is already a bottleneck resulting from event reporting. However, memory protection is not needed for a MC to forward pseudoevents to EMs. It is only needed once for the context switch between TP and the MC. The Alamo framework provides a special event code E_MemViol that monitors can use to indicate enabling or disabling the memory protection feature. This gives EM writers flexibility of turning on and off this feature at run-time. The performance impact of enabling memory protection on MC and EMs is presented in Chapter 6.

2.4 Symbol table relocation

Relocation is a process of connecting symbolic references with symbolic definitions. For example, when a program calls a function, the associated call instruction must transfer control to the proper destination address at execution. A relocatable object file contains information that describes how to modify its section contents, thus allowing the executable file to hold the right information for a process's program image.

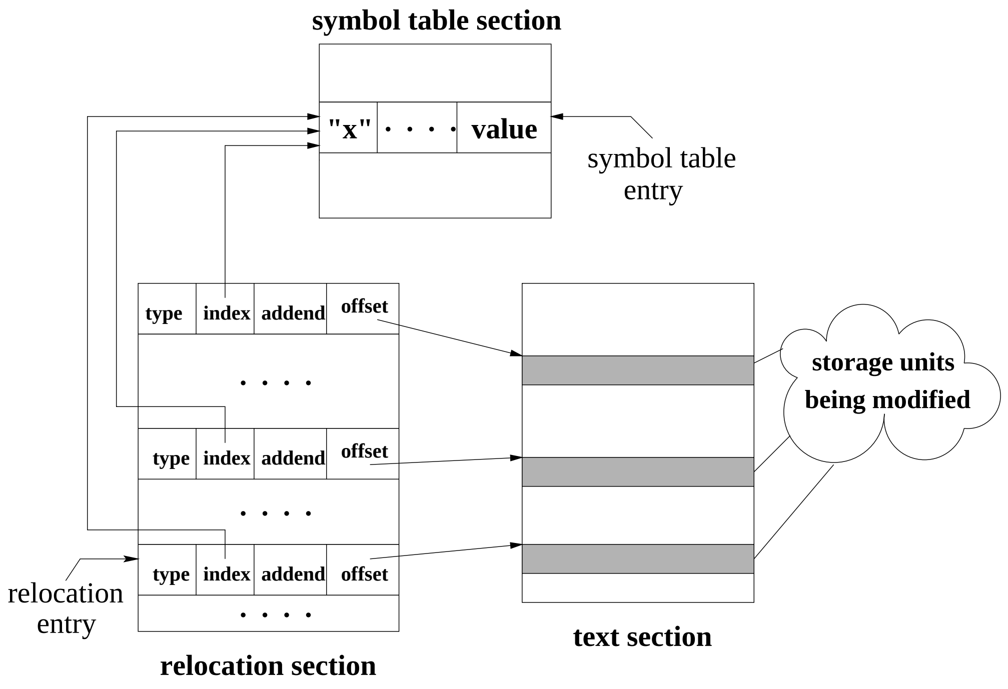


Figure 2.8: Relocation of symbols defined within the program using the information provided by relocation and symbol table sections in ELF.

Figure 2.8 shows the relationship between a relocation section, the symbol table section, and a section being modified in an ELF object file. It represents an example of how to relocate the global variable x in the loaded program using the information provided by the relocation and symbol table sections. A relocation section in the ELF object file is used to hold information of how to modify its section contents. Its section header references two other sections: an associated symbol table section and a section to which the relocation applies. The indices for those two sections are specified by the two member fields from the section header structure. The relocation section contains several relocation entries. Each relocation entry designates the information that is needed to modify the contents of a specified storage unit location. Detailed descriptions of the symbol table entry and relocation entry structures can be found in [Elf93], pages 120 – 130. A relocation type specifies which bits to change and how to calculate the actual address for a symbol. The number of relocation types used for relocating global symbols depends on the machine level instructions and the addressing modes they use. It is important to point out that, as shown in Figure 2.8, if symbol x is referenced several times within a function, then there is one or more relocation entries associated with each reference of x within that function. Values of r_addend and r_offset from the relocation entry structure, or st_value from the symbol table entry structure are used with arithmetic and bit operations to calculate the actual address for the symbol x.

Several issues arise from calculating the address to be stored into the modified region. They are listed as follows:

The r_offset member from the relocation entry structure designates the offset from the beginning of the section to which the relocation action applies. In the calculation of the value to be stored, the absolute address for the location at which the relocation applies is used instead. Since the AME loads the program, the address of each section in that object file is known by the AME. The absolute address is calculated by adding the starting address of the section being modified with the offset value specified by r_offset.

The st_value member from the symbol table entry structure has slightly different interpretations for different types of symbols. In relocatable object files, st_value holds a section offset for a defined symbol and the alignment constraint for a symbol whose section labels a common block that has not yet been allocated. Figure 2.9 shows the relationship between a symbol table section, a string table section, and the section a symbol (*e.g.* x) resides in. The string table section holds null-terminated character sequences representing symbol and section names. For those defined symbols, AME updates its st_value field to the absolute address by adding the starting address of the section it resides in. For those symbols that belong to a common block, AME allocates space during the loading process. The absolute address of those symbols is assigned to their st_value field.

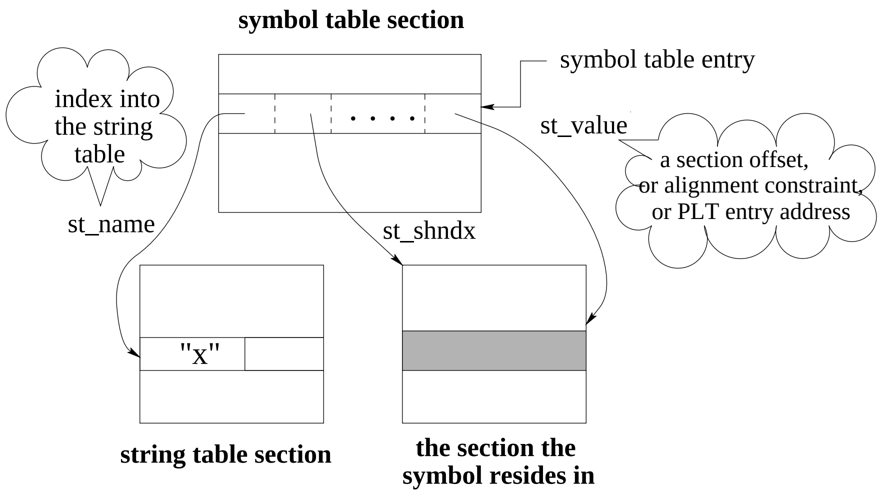

symbol table section
symbol table entry
index into the string table
st_value
a section offset, or alignment constraint, or PLT entry address
st_name
st_shndx
"x"
string table section
the section the symbol resides in

Figure 2.9: The relationship between a symbol table section, a string table section, and the section a symbol (*e.g.* x) resides in.

Because shared library codes are not statically linked in with the loaded program, the shared library symbol's st_value and st_shndx values are both zero indicating entries are not valid for those variables. As shown in Figure 2.10, both the ELF object file for a loaded program and the executable file for the AME have a symbol table section. The st_value field for shared library symbols in the executable file contains the address of its

PLT entry location. The AME updates those invalid entries in the ELF object file to their corresponding PLT entry locations in the AME program image. As mentioned in the earlier section, the PLT entry for shared library symbols contains a jump instruction. The jump instruction transfers control to the system's dynamic linker to find the symbol's real address, and transfer control to the desired destination.

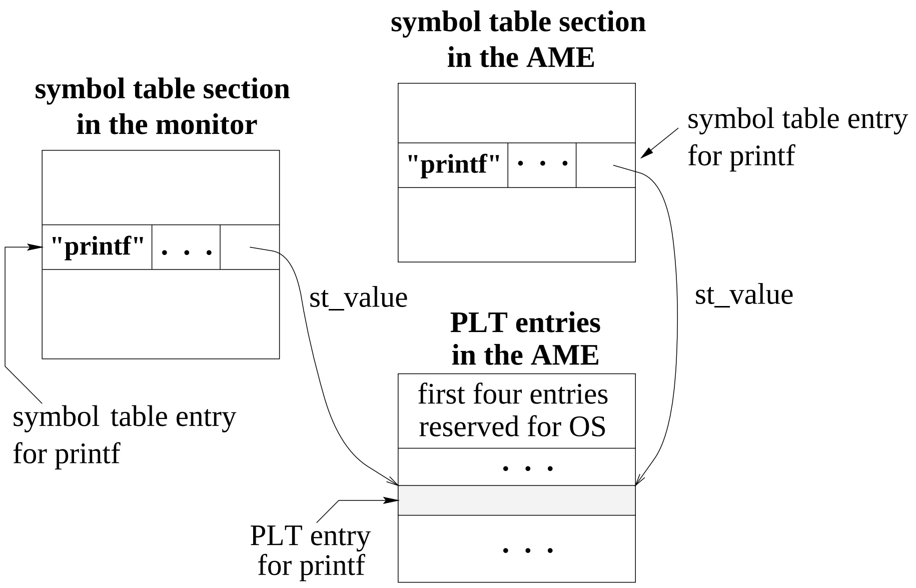

symbol table section in the AME
symbol table section in the monitor
symbol table entry for printf
"printf"
. . .
"printf"
. . .
st_value
st_value
PLT entries in the AME
first four entries reserved for OS
. . .
. . .
symbol table entry for printf
PLT entry for printf

Figure 2.10: A diagram showing how to relocate shared library symbols in the loaded program using PLT entries in the AME executable program image.

As an example of a relocation type: R_SPARC_WDISP30, the calculation specified in [Elf93] for this type is: $(S + A - P) \gg 2$. It represents the modification of a 30-bit data at the target location. S denotes the absolute address of the symbol, which is the modified st_value field for that symbol. A is the r_addend field in the relocation entry structure. P is the absolute address of the storage unit being relocated, which is the modified r_offset field in the relocation entry structure for that symbol.

Chapter 3

AME Coroutine Model

The execution model employed by AME is a coroutine model [Marl80]. *Coroutines* provide an attractive model for writing execution monitors. The coroutine model is specified in terms of a procedure call in which the values of local variables are retained even when control is not within that process [Jeff93]. It is more flexible than a procedure call based model in that monitors can have their own control flow logic, but it possesses similar properties to a procedure call in that when the control flow is transferred to another process, the state of the current process is saved. Upon re-entering the suspended process, the execution continues from the point where control was transferred.

The performance of the context switch from TP to monitor and back is critical because of the high number of events (millions or billions) processed during monitoring. The actual implementation of coroutines has substantial impact on the feasibility of this framework. AME initially used Solaris threads to implement coroutines and semaphore operations to suspend and resume them, allowing only one thread to be active at a time. Subsequently, AME was modified to employ a simpler coroutine model based on Icon's co-expression data type [Gris90].

This chapter first presents AME's context switch mechanism in detail, followed by the mechanisms of event reporting and filtering that are used to further minimize the overhead associated with monitoring. The synchronization and coordination of several EMs is discussed at the end of the chapter.

3.1 AME context switch

TP and EMs are separately loaded entities within the AME. EMs use an *event mask* to explicitly specify what kind of events are to be reported. The event mask is a bit vector with each bit representing a particular event code. When a bit is set, it means the event code corresponding to that bit is used by an EM. Whenever the TP encounters an event specified in the event mask, execution control is transferred to the EM along with a code and a value for that event. When this occurs, an event has been *reported* to the EM. When an EM resumes the execution of the TP, it can explicitly specify and change the event mask. The nature of event reporting and masking is discussed further in Section 3.2. Figure 3.1 shows the flow chart of the event generation and control transfers between TP and EM for a single event report. As the program is running, events are generated at various points. They are shown as tick marks along the time line. Not every event is reported to the EM.

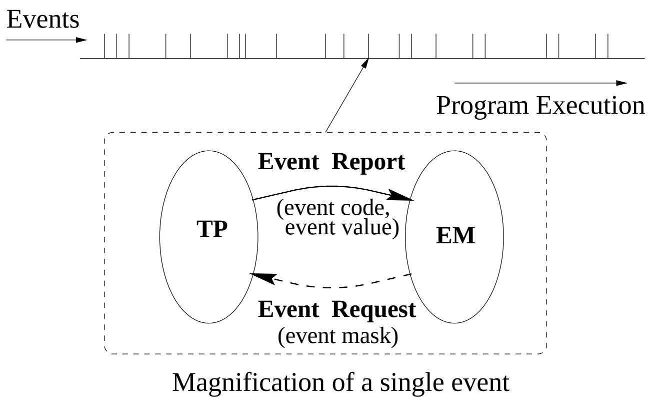


Figure 3.1: Flow chart of the event generation and control transfers between TP and EM

The first approach to implement the context switch in AME was to use the Solaris threads package. Under Solaris threads, AME assigns each loaded program a separate thread, but because of the coroutine model, at any instant only one thread is running. Semaphores are used to synchronize the execution of multiple threads. Usually threads are used for concurrent programs to explore the advantage of parallelism between threads. However this feature is not needed in the current framework. Since the execution model is not concurrent, and semaphore operations impose additional overhead, we choose the second approach because it removes the semaphores and threads and uses a more portable and simpler model.

Our second approach is based on the activation of co-expression implemented in the Icon language [Wamp81]. A *co-expression* in Icon is a data object that contains a reference to an expression and an environment for the evaluation of that expression [Gris90]. Activation of one co-expression by another co-expression entails that evaluation is interrupted in the first and continued in the second. This mechanism is adopted by our framework. In our second approach, a new user-level context is created for each loaded program. The new context includes a stack region allocated out of the heap space of that program. Each program has a handle to its context. Each context executes with its own control flow. It uses its own stack, and a set of registers. On the Sparc, the stack must be aligned on the correct boundary. Register values including sp, fp, and pc are saved onto the stack before the control is transferred to another context. The first time the control for a new context is activated, the function call to execute that new context is performed. The execution entry point for that context is passed in from the AME. For subsequent invocations of the same context, the execution just resumes from where it left off. A single copy of a global variable which represents the current executing context is maintained for both TP and EMs. Each time a context switch occurs, the appropriate global variable is updated. The core piece of code for the context switch is written in assembly code and has been ported to many different machines and operating systems, such as UNIX, MS-DOS, Macintosh, VMS, *etc.* Thus this approach is more efficient timewise and more portable than the Solaris thread approach. It is necessary to mention that this core piece of code is obtained from the earlier implementation for the Icon co-expression.

3.2 Run-time event reporting and masking

Since Alamo can provide extremely fine-grained events, the number of events that occurs during a program execution can be extremely large. It is large enough to create serious performance problems in a real interactive system. Most EMs function effectively on a small fraction of the available event set. Efficient support for the selection of appropriate events to report and the minimization of the number of event reports are primary concerns.

AME supports dynamic event masking based on event codes. It chooses to mask events in the TP instead of filtering them in the EM. This reduces the number of context switches between the TP and EMs. Moreover, event masking allows an EM to specify what events are to be reported and to change the specification at run-time. when the TP starts execution, the EM selects a subset of possible events from which to receive its first report. The TP executes until an event occurs with a selected code, at which point it reports the event to the EM and suspends itself. After the EM has finished processing the report, it resumes the TP and specifies an event mask. Dynamic event masking allows the EM to change the event mask in between event reports.

A macro EV(event, value) is used by TP for reporting events. *Filtering* which refers to run-time selection of those events that are used and those discarded by a monitor is performed by this macro. The TP first tests the event code with the event mask. If the bit is set (or 1), the actual event report occurs. Otherwise, no event is reported. Since the filtering is done on the TP's side, it minimizes the number of event reports.

The primary test for whether an event is of interest is performed in-line with no context switch to the monitor.

This macro also performs the task of fetching the current target program's program counter, stack pointer, and frame pointer values. Those values are important for monitor writers to obtain information about which variable they are accessing. For example the rvalue, y, in the statement x = y; has an associated variable dereference event. For each dereference event the event value is the address of that variable. Monitor writers can use the provided function: int EvStab(int addr, struct evstab *stab); to find out the name of the variable being accessed from the address given, using the current pc, sp, and fp values. The function fills in a structure evstab buffer with the actual pointer to the variable name in the string section and the descriptor value of this variable. This function returns non-zero indicating error. The detailed description of the structure evstab can be found in section 5.3.2. Chapter 4 gives more detailed information about the TP state access library's functionalities.

Events are requested by an EM using the function EvGet(mask). EvGet() activates the context of the program from which the EM gets its events. It also passes the information about who is monitoring the TP and what is the current event mask for this EM to the TP. The TP continues execution until an event report takes place.

3.3 Monitor coordination

AME's dynamic loading capability allows simultaneous execution of multiple EMs and a single TP. This allows users to write simple special purpose EMs quite rapidly. The difficulty posed by multiple monitors is not only in loading the programs, but also in coordinating and transferring control among those monitors and providing each EM with the TP execution information it requires. Monitor coordinators (MCs) are thus used to perform the synchronization and provide monitoring services to additional EMs.

Figure 3.2 [Jeff93] shows some configurations for multiple EMs with a TP and a MC. The tree structure layout represents the parental event report relationship. In the case when there is a MC in between TP and EMs, the MC must be able to forward events to desired EMs. EMs receive events from the MC as if they are from the TP. Figure 3.2(a) shows an EM monitoring a TP without a MC, which is a typical case. Figure 3.2(b) shows a MC is used for an EM monitoring a TP. The MC is just an execution monitor that forwards events. Figure 3.2(c) shows the main purpose of using a MC, the execution of multiple EMs on a single TP.

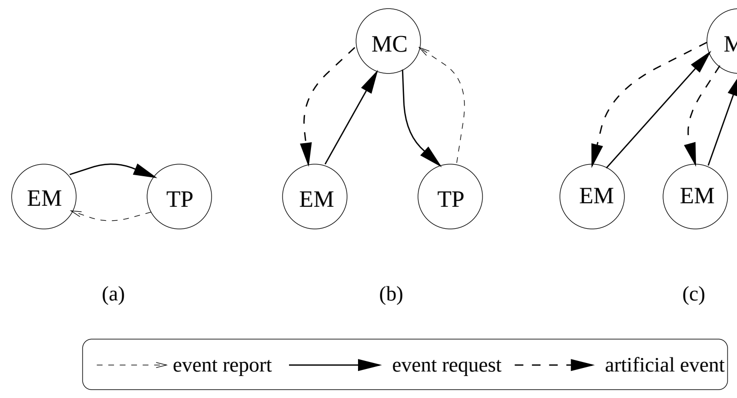


Figure 3.2: Several configurations using a MC. (from [Jeff93])

3.4 Advantages and disadvantages of the MC approach

Since Alamo's design is based on an execution monitoring framework developed for the Icon programming language [Jeff94] and generalizes that model to encompass compiled, systems programming languages, the three major advantages of using MCs in the execution monitoring are similar to those described in [Jeff93]. They are modularity, specialization, and extensibility.

Modularity *With a MC, monitors can be developed independently of one another and of the MC itself; they can be individually loaded by the AME with the program to be monitored.*

Specialization *Support for multiple monitors allows EMs to be written to observe very specific program behavior and still be used in a more general setting. This in turn reduces the burden of generality placed on EM authors. Specialization also simplifies the task of presenting information.*

Extensibility *Extensibility refers to the ease with which new tools are added to the visualization environment. Adding a new tool to run under a MC does not require recompiling or even relinking the MC or any of the other visualization tools.*

The monitor coordinator approach on the other hand has its disadvantages. Using MCs introduces some additional context switches among programs. This in turn can degrade the performance of the program. Although unsuitable for monitor development, a single monolithic EM can perform better than a MC with multiple EMs. For this reason, if there is only one monitor monitoring the TP, an MC is not required. This removes unnecessary context switches and results in better performance.

Chapter 4

TP State Access

As described in Chapter 1, AME allows execution monitors (EMs) to gather extensive information from an executing target program (TP), starting with the information from instrumented events. To acquire further information, EMs require the ability to directly access the TP's variables and structure information while processing the events. This functionality enables the EM writer to extract more information from the running target program and understand the program behavior better.

EM access to TP state is provided by several means. Since EMs do not have compile-time knowledge of TP types, a generic value datatype and related service functions implement a run-time traversal mechanism for TP values including arrays, structures, and unions. EM global variables corresponding to critical pieces of TP state are automatically assigned during event transmission. EM library functions provide access to TP symbol tables, scopes, addresses and values of variables, and stack traversal.

This chapter begins with the discussion of the decoding of debugging symbol table information. This information is produced by the assembler and linker with the **-g** option of the GNU C compiler. The design and programming interface of the AME used by monitor authors are then presented.

4.1 Debugging symbol table information

Stabs refers to a format for information that describes a program to a debugger. This format was invented by Peter Kessler at the University of California at Berkeley [Stab93]. It is the native format for debugging information in the a.out and XCOFF object file formats. The GNU C compiler emits stabs format debugging information in the ELF object files. This debugging information describes features of the source file such as the line number, the types and scopes of variables, and the names, parameters, and scopes of functions. Decoding this information enables EM writers to peek into the TP at run time. In the ELF object file, debugging information is stored in .stabstr and .stab sections. The following subsections will discuss them in detail.

4.1.1 .stabstr section in the ELFs

The .stabstr section contains the string field for the stabs. It holds the meat of the debugging information. The flexible nature of this field is what makes stabs extensible.

The format of the string field is *"name:symbol-descriptor type-information"*. *name* is the name of the symbol represented by the stab. It can be omitted, which means the stab represents an unnamed object. The *symbol-descriptor* following the ':' is an alphabetic character that tells more specifically what kind of symbol the stab represents. If the symbol-descriptor is omitted, but type information follows, then the stab represents a local variable. For a list of symbol descriptors, see [Stab93]. *type-information* is either a *type-number*, or '*type-number=*'. A *type-number* alone is a type reference, referring directly to a type that has already been defined.

The '*type-number=*' form is a type definition, where the number represents a new type which is about to be defined. The type definition may refer to other types by number, and those type numbers may be followed by '=' and nested definitions. In a type definition, if the character that follows the equal sign is non-numeric, then it is a *type-*

descriptor, and tells what kind of type is about to be defined. Any other values following the type-descriptor vary, depending on the type-descriptor. The list of type-descriptors is also available in [Stab93].

This string field gives the type information for all the variables in a program. Due to the nested nature of type definitions, the string field can be quite complicated. A simplified grammar for decoding the string field for a C program is presented in Appendix B. AME uses this grammar to build its own symbol table for variables in the TP, which contains the type information for them.

The grammar adopted by the GNU C compiler represents the rules of defining different types in C. Types can be categorized into builtin types, subrange types, array types, structure or union types, enumeration types, and function types. Below are some examples showing what the string field looks like for simple type definitions.
The first example is for defining builtin types (int, char, void, float, *etc.*):

```
int:t1=r1;−2147483648;2147483647;
char:t2=r2;0;127;
...
signed char:t10=r1;−128;127;
unsigned char:t11=r1;0;255;
float:t12=r1;4;0;
double:t13=r1;8;0;
void:t19=19
...
```

The symbol-descriptor ‘t’ following ‘:’ indicates that it is giving a type a name. In the above case, it defines type 1 is int and type 2 is char. The GNU’s C compiler uses subranges of themselves to define traditional integer types (*e.g.*: int and char). For subrange types, it uses ‘r’ type-descriptor to indicate it is a subrange of another type. Type-descriptor ‘r’ is followed by type information for the type of which it is a subrange, a semicolon, an integral lower bound, a semicolon, an integral upper bound, and a semicolon. GCC also uses subranges of int to describe other builtin types (*e.g.:*

unsigned char, signed char, float, *etc.*). If the upper bound of a subrange is 0 and the lower bound is positive, this indicates the type is a floating point type, and the lower bound of the subranges indicates the number of bytes in the type (*e.g.:* float, double, *etc.*) as shown in the above example.
The second example is for defining array types. For example, the definition:

```
char char_vec[3] = {'a', 'b', 'c'};
```

produces the output:

```
char_vec:G19=ar1;0;2;2
```

The symbol-descriptor 'G' following ':' indicates that char_vec is a global variable. Array types use the 'a' type-descriptor. Following the type-descriptor is the type of the index and the type of the array elements. In the above example, the type of the index is represented by 'r1;0;2;'. It defines the index type which is a subrange of type 1 (integer), with a lower bound of 0 and an upper bound of 2. This defines the valid range of subscripts of a three-element C array. The type of the array elements is type 2 (char).

For defining more complicated types, please refer to the reference [Stab93].

4.1.2 .stab section in the ELFs

The .stab section encodes the structure of the program. The elements of the program structure it encodes include the names of functions, the parameter information of a function, global, local, and static variables, the line number information, and the beginnings and ends of a scope. Decoding of this section gives EM writers a detailed structural view of a TP.

The format of an entry for this section is as follows:

```
struct internal_stab {
    long stab_strx; /* index into string table of name */
    unsigned char stab_type; /* type of symbol */
    char stab_other; /* misc info ( usually empty ) */
    short stab_desc; /* description field */
    unsigned long stab_value; /* value of symbol */
};
```

The stab_strx field holds the offset in bytes of the string within the .stabstr section. The stab_type field specifies different types of the stab. The full list of stab types can be found in [Stab93]. In most cases, the stab_desc field contains the line number information for functions or variables. The stab_value field contains the offset from the frame pointer of a function for automatic variables and parameters of that function. It is negative for automatic variables and positive for parameters. It can contain the number of the register where the variable data will be stored for register variables. But for global and static variables this field contains useless information, since it needs to be relocated in ELF.

Figure 4.1 shows the layout of the .stab section for the program shown on the right side. The stab representing a procedure name is directly followed by a group of other stabs describing the procedure's parameters, its local variables, and its block structures.

As shown in the Figure 4.1, the program's block structure is represented by the start (left curly brace) and end (right curly brace) stab types. The variables defined inside a block precede the start stab type for the GCC compiler. The start and end stabs that describe the block scope of a procedure are located after the function name stab that represents the procedure itself. The value field of the start and end stabs specifies the start and end address of that code block, respectively. They are relative to the function in which they occur.

The line number information stabs located in between the stabs describing the function's parameter and the function's local variables. The stab_desc field contains the line number and the stab_value field contains the code address for the start of that source line. In the ELF, it is relative to the start of the function.

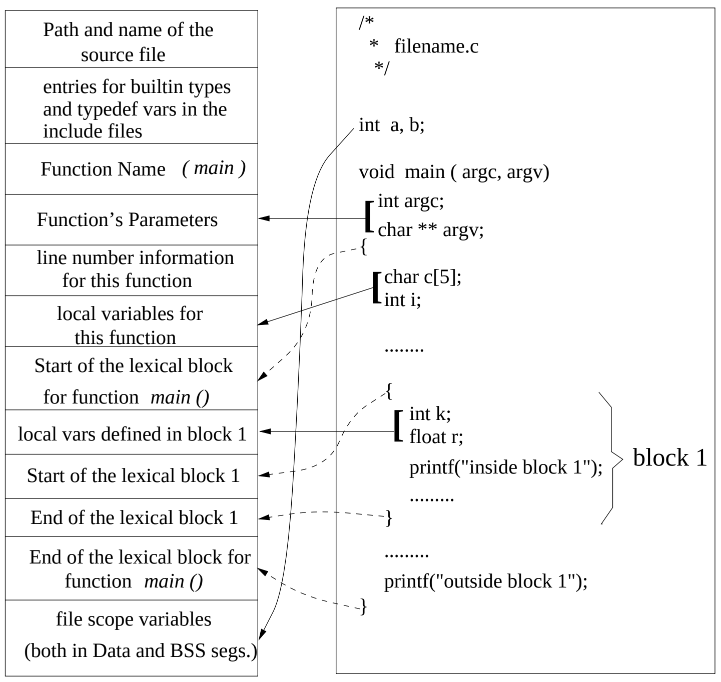

Path and name of the source file
entries for builtin types and typedef vars in the include files
Function Name (main)
Function's Parameters
line number information for this function
local variables for this function
Start of the lexical block for function main ()
local vars defined in block 1
Start of the lexical block 1
End of the lexical block 1
End of the lexical block for function main ()
file scope variables (both in Data and BSS segs.)
/*
* filename.c
*/
int a, b;
void main (argc, argv)
int argc;
char ** argv;
{
char c[5];
int i;
....
{
int k;
float r;
printf("inside block 1");
....
}
....
printf("outside block 1");
}
block 1

Figure 4.1: A sample layout of the .stab section for a program.

Also as shown in the Figure 4.1, the stabs describing global variables are located at the end of the .stab section. They are defined outside any block structures and are grouped together at the end.

Further information about the stabs can be found in [Stab93]. As a conclusion, combining the information from .stabstr and .stab sections, the type and scope information for variables in the TP can be obtained. The following sections describe the library functions provided to the EM writer for accessing TP's state conveniently.

4.2 Descriptors

In AME, EMs use *descriptor*s to refer to TP variables. Descriptors include type information from the debugging symbol table sections in the TP object module. The descriptor structure is as follows:

```
typedef struct {
    struct ctype *type;
    void * addr;
} Desc;
```

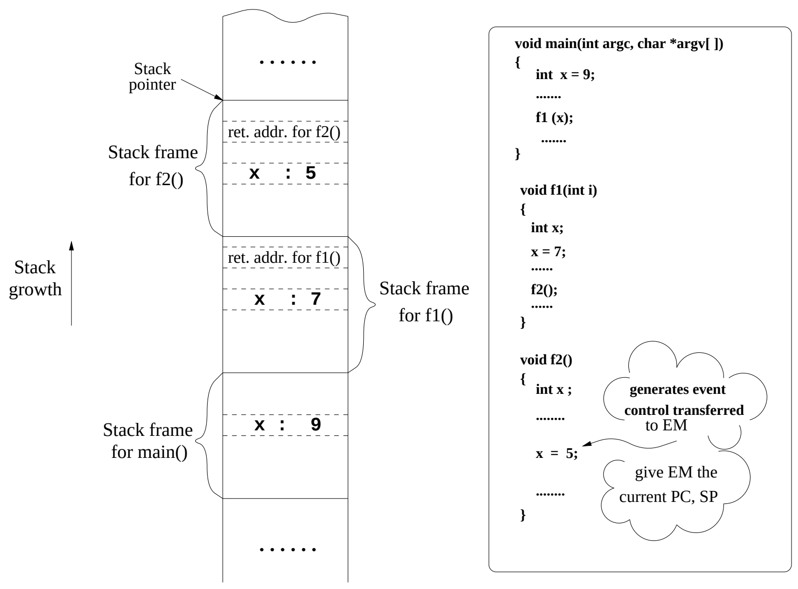
Descriptors have two fields: type is a pointer to a structure that contains the type information. It is not used directly by EM authors; instead, service functions use this pointer to extract type information for a given variable. addr is a void pointer that holds the memory location where the variable is stored.

4.3 Target program's stack frame

As mentioned earlier in Chapter 3, the TP saves its registers and automatic variables onto its stack before the control of program execution is transferred to EMs. Figure 4.2 shows the TP's stack when the control is transferred to the EM for the program shown on the right side of the diagram. The main() function calls function f1(). Inside f1(), it calls another function f2(). At some line in the f2(), the event is generated by CCI and this is the point where the control is transferred to the EM. Service functions are provided for EM authors to walk up and down the TP's stack to inspect any particular stack frame of interest. These service functions use several global variables internally to track the stack frame that is currently being visited. EM authors can inspect the current stack frame and retrieve information, such as values, types, or addresses of variables from it.

4.3.1 EM global descriptor variables

EMs have several global descriptors: Params, Locals, Local_Statics, Globals, and Statics. These descriptors are used directly by EM authors to get handles for the parameters, local variables, and static variables for the function being visited, global variables, and file scope static variables, respectively. They are accessed using the same notation that is



```
void main(int argc, char *argv[ ])
{
   int  x = 9;
   .......
   f1 (x);
    .......
}

 void f1(int i)
 {
   int x;
   x = 7;
   ......
   f2();
   ......
 }

 void f2()
 {
   int x ;
   ........
   x  =  5;
   ........
 }
```

Figure 4.2: A view of the TP's stack frame when the control is transferred to the EM.

used for target program struct values. The type field of the descriptors for these implicit structures contains the program counter value for the current stack frame, and the addr field contains the frame pointer value for the current stack frame.

C does not allow nested functions, but does allow nested block structures called compound statements inside pairs of curly braces. Variables declared inside each pair of curly braces are in a new scope that is one level deeper than the variables declared outside. The scope is a conceptual term for distinguishing the visibility of variables inside one file or between several files. For each function that is still active, there is a stack frame (or activation record) associated with it as shown in Figure 4.2. We can have several scopes within each function and all the local variables declared in each scope will be stored in the stack frame if they are alive. The Locals represents the local variables for the function being visited at the scope which is indicated by the program counter value. An example C program is shown in Figure 4.3:

For the program shown in Figure 4.3, when the control is transferred at the

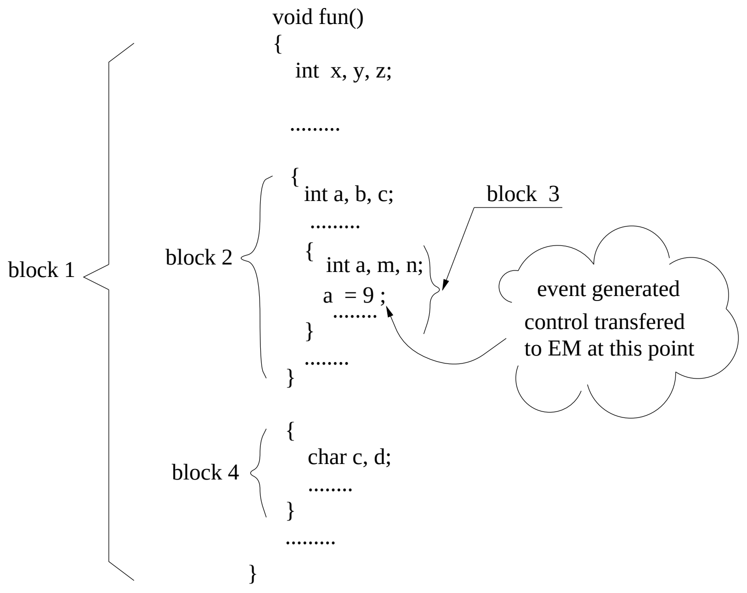


Figure 4.3: An example C function with several block structures defined inside it.

assignment a = 9, Locals in EM refers to the variables declared inside block 3. Block 2 and block 1 are direct ancestors of block 3. Block 1 is also a direct ancestor of block 4. Block 2 and block 4 are siblings. When the program is inside block 4, it cannot reference variables declared inside block 2, since the scope associated with block 4 is outside the scope of block 2.

4.3.2 Traversing the TP's stack and accessing named variables

In order to access local variables in different invocations of functions in the target program, EMs must be able to traverse the TP's stack. There are two functions provided for moving the internal frame pointer and program counter that keep track of the stack frame being visited:

- int UpStack();

  This function moves the EM's view of the TP stack "up" to the previous function call's stack frame. It returns 1 upon success, or 0 if the current frame is on the TP's main() function.

- int DownStack();

  This function moves EM's view of the TP stack "down" one stack frame. It returns 1 upon success, or 0 if the current frame is on the frame where the TP was suspended.

- Desc EvVar(char *name);

  Given the name of a variable, this function first looks up the symbol in the local symbol table for the current stack frame in the scope indicated by the program counter. If it is not found, the local symbol tables that are the direct ancestors of the current scope are searched in sequence. If it is still not found, the global symbol table is searched. This function returns a descriptor with the type and address where this variable is stored in memory. If the symbol is not found in any of the symbol tables, this function returns a descriptor with both the addr and type fields being NULL indicating an error.

Using these three functions, an EM author can obtain information for any given named variable in the current stack frame where the internal frame pointer and program counter are at.

4.4 Functions for accessing TP's state information

4.4.1 Accessing component values using descriptors

EMs do not in general know the names, or types, of TP variables and their components, such as struct or union fields, or array elements. Another set of functions produces the names and types of component values. The functions take a descriptor parameter. The descriptor can either be an implicit structure, such as Locals, Params, Local_Statics, Globals, Statics, or it can be an ordinary C structure, union, or array value. Below is a brief description of various functions that extract component information from a descriptor.

- int EvNum(Desc var);

  This function returns the number of components described by the descriptor var.

- Desc EvElem(Desc var, int i);

  This function returns a descriptor for the ith component described by the descriptor var.

- char *EvName(Desc var, int i);

  This function returns the name of the ith component described by the descriptor var.

For example, using the implicit structures as the parameter for the above functions:

```
int i;
Desc temp;
char *name;
/*
 * get the number of local variables for the current scope
 */
i = EvNum(Locals);
/*
 * get the 2nd parameter for the current function
 */
temp = EvElem(Params, 2);
/*
 * get the name for the 2nd global variable
 */
name = EvName(Globals, 2);
```

In addition to implicit structures, a descriptor can represent an ordinary C value and its type. In the above example, temp which retrieves the 2nd parameter for the current function may be a structure itself. The above functions can also be used to extract similar information from an ordinary C value. The code sample would be:

```
int i;
Desc d;
Desc temp;
char *string;
/*
 * extracts the 2nd parameter, which is a structure, and places it in
 * the descriptor temp.
 */
temp = EvElem(Params, 2);
/*
 * i contains the number of fields of the structure.
 */
i = EvNum(temp);
/*
 * d holds the descriptor for the 2nd field of the structure
 */
d = EvElem(temp,2);
/*
 * string gets the name of the 3rd field of the structure
 */
string = EvName(temp,3);
```

The above three functions enable EMs to access variable information in the current stack frame of the program indicated by the internal frame pointer value. In some cases, the EM writer would need to access a variable that is hidden by the current scope. As an example shown in Figure 4.3, when the target program gives control to EM, EM wants to access the variable a which is declared in block 2. The function: Desc WhereIs(char *name, int i); returns a descriptor for the variable named name in the scope i levels outside the current scope. When i is 0, this is the current scope. When i is 1, this is the immediately enclosing scope, and so on. The code example would be:

```
Desc d1, d2, d3;
/*
 * d1 holds the descriptor for the variable ”a” in block 3.
 */
d1 = WhereIs(”a”, 0);
/*
 * d2 holds the descriptor for the variable ”a” in block 2.
 */
d2 = WhereIs(”a”, 1);
/*
 * d3 holds the descriptor for the variable ”a” in block 1,
 * which should be a NULL descriptor value indicating not found.
 */
d3 = WhereIs(”a”, 2);
```

Note that a local variable is alive only if it is declared in a direct ancestor of the current scope. EM authors cannot ask for information about dead local variables, since their memory is reused by other variables.

4.4.2 Type, value, address, size, and location information

There are several functions on descriptors that give EM writers the ability to obtain information about the size of a variable, the type of a variable, and the value obtained by dereferencing a variable. A function which provides the current program execution's location information including the line number and file name is also provided among the EM library functions. The detailed description of those functions can be found in Appendix A.

4.4.3 Predicates

Predicate functions give EM authors a means of comparing types and values of different descriptors conveniently. It also gives the author the ability to gather information about the membership of a compound data type (structure, union, or a function) which a descriptor represents. Those predicate functions return `1` for true, `0` for false. They are also listed in Appendix A.

Chapter 5

Sample Monitors

The Alamo framework provides a variety of information about a program execution. For example, it supports the development of monitors for the memory allocation behavior of a program, the control flow of a program, data structure accesses, file I/O operations, and communication activities of a program. This chapter presents simple example execution monitors that demonstrate the viability of the framework and show how Alamo is used to perform typical monitoring tasks. The examples explore two major categories of applications for this framework, profiling and memory monitoring.

Profiling tools can be used to monitor the number of function calls, floating point operations, and branches within a program. In general, profiler information gives the user an idea about how often the program executes a particular part of code. Users can then rewrite the most frequently executed parts to run more efficiently. Memory monitoring tools can be used to detect memory leaks, illegal memory references, *etc.*

All Alamo monitors follow a common outline, described by the template in the next section. Following that, sample execution monitors for each category of the application are given. The examples are actual program fragments that demonstrate various methods for monitoring using the Alamo framework.

5.1 Template for an EM

A typical EM follows the general outline shown below: It first sets up the execution monitoring by calling the procedure EvInit(argc, argv). Then the initial event mask for the EM is created using the procedure EvMask(n, ...). EvMask(n, ...) builds a mask with n event codes in subsequent parameters. The main body of the EM consists of an infinite loop that requests and processes events from the TP. After an end of execution event with the event code END_OF_EXEC is received, the EM calls the procedure EvTerm() to terminate the monitoring and exits the program.

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

This template is generally omitted from the following code examples for the sake of brevity. These examples are plugged into the template while loop at the point commented by “process events”.

5.2 Profiling tools

Events generated by CCI [Temp96], Alamo's source-language level instrumentation tool, can depict low-level behavior, such as unary or binary operators and variable accesses. They can also correspond to higher-level behavior, such as function calls, loops, and control structures. Since events are generated as the program executes, EMs can profile various kinds of program activity at run-time. This is different from traditional profilers which are post-mortem analysis tools. Alamo profilers can give exact counts and cross-reference source-code location information, unlike traditional profilers that are based on statistical approximations. In addition, Alamo profilers can work at a finer detail level (expression-level) than most profilers which work at a procedural level.

For example, using the Alamo framework, an EM can profile the number of floating point operations, the number of times a branch is taken for the else part of an if statement, or the number of iterations executed by a while loop. By cross-referencing other events with program location information, or at a higher level by means of a configuration file, an EM can also profile the number of times a particular function is called.

A simple EM which profiles floating point operations is shown below. It increments counters for different floating point operations as the program runs. This approach can be used to profile any events as long as the event mask is set to collect those events.

```
switch(eventcode) {
    case E_Plusf:
        Plus_cnt ++; break;
    case E_Minusf:
        Minus_cnt ++; break;
    case E_Multf:
        Mult_cnt ++; break;
    case E_Divf:
        Div_cnt ++; break;
}
```

These kind of tools can assist in optimizing performance by revealing potential places where the user can tune the program. For example, a large number of floating point division operations may indicate a possible area of improvement. By cross-referencing other events such as assignment events, loop control events, and source-code location events, the user can identify the positions of particular bottlenecks. Given such locations, a programmer may be able to remove some redundant operations and speed up the program execution.

Besides the simple counting of a certain event, more elaborate tools such as animating the control structures and flow of a program can be developed with the aid of graphics techniques. The "flow graph" of a program can be constructed at run-time by tracking where the program location goes (where it loops, where it branches, where it enters or leaves a function). By feeding different inputs into several executions of an application, the "flow graph" tool can mark paths that have been executed or highlight paths that have not been executed.

5.3 Memory monitoring tools

Memory usage is an important aspect of program behavior that is not directly evident from the source code examination. Memory references can be categorized as referencing global section, stack and heap regions. Most memory bugs occur when programmers manipulate memory in the heap region. Also there are common bugs associated with illegal memory references for all kinds of memory regions. The following sections present sample monitors for detecting memory leaks and for checking array subscripts to detect out of bounds references.

5.3.1 Memory leaks

From a set of primitive events that are provided for all function calls, namely E_Call, E_Param, and E_Ret, CCI can be configured to produce several events for every memory allocation which uses functions malloc, calloc, *etc.* For example, it can generate a memory allocation event with an event code E_Malloc and a corresponding event value which gives the address of the memory allocated. In addition, CCI can be configured to generate an E_Free event when the function free is called. Its corresponding event value specifies the address of the object being freed. Using these events, an EM writer can easily write monitors to detect memory leaks in a program. The EM sets its event mask to collect E_Malloc and E_Free events.

The following example uses a set data type (omitted here) with insert() and delete() operations to maintain the addresses of allocated objects in a program. Every time an E_Malloc event occurs, the address of the allocated object is inserted into the set, whenever an E_Free event occurs, the corresponding object is deleted from the set. At the end of the program execution, if the block_set is not empty, there is a memory leak.

```
switch(eventcode) {
    case E_Malloc:
        insert(block_set, eventvalue);
        break;
    case E_Free:
        delete(block_set, eventvalue);
        break;
    ...
}
```

More sophisticated monitors can also track variable names which are pointing to those allocated objects and give users more information about which variables are not freed or do other kinds of memory checking for bad behavior, such as freeing a block that has not been allocated, or freeing a block twice.

5.3.2 Array index checks

The instrumentation also produces events for structure field and array element accesses. Another common bug is trying to access an array element that is out of bounds. A simple EM that detects this problem demonstrates that the access functions provided by the framework extend the capabilities of monitors in Alamo well beyond what is provided by the event model. In the event model, the TP is instrumented to report events that EMs ask for. The access function library complements the event-based communication from EMs to the TP. It enables EMs to peek into the TP and obtain state and variable structure information at run-time. The variables used in this example are shown below.

```
int eventcode;
struct evstab arstack[MAX_STACK_DEPTH];
int level=0;
Desc elemdesc;
```

arstack is an array of evstab structures which hold the name and the descriptor (described in Chapter 4) for array variables. The evstab structure has two fields, name and desc. arstack is used as a stab stack to store variable names and descriptor information for the multi-level deep array references, since several array reference events may occur before the corresponding index events are resolved. The stack depth for the array reference is seldom more than five levels deep. For example, the event sequence generated for array1[array2[array3[index]]] is shown in the Figure 5.1:

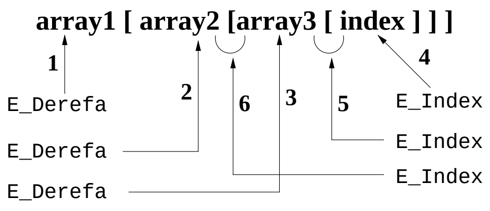


Figure 5.1: The array reference event sequence generated by the expression array1[array2[array3[index]]].

The event value for an E_Derefa event is the address of the array being accessed. The event value for an E_Index event is the integer value of the array index. elemdesc is a descriptor for the array element being accessed. The code used for detecting the array out of bounds access is shown below:

```
switch (eventcode) {
    case E_Derefa:
        EvStab(eventvalue, &(arstack[level++]));
        break;
    case E_Index:
        elemdesc = EvElem(arstack[–level].desc, eventvalue);
        if ( IsNull(elemdesc) )
            fprintf(stderr,
                ”array access out of bounds: (%s)[%d]\n”,
                 arstack[level].name, eventvalue);
        break;
}
```

Function EvStab fills in the evstab structure passed in as the parameter with the actual pointer to the array's name in the string section and the variable's descriptor value. The name of the array variable and its descriptor value are pushed onto the stab stack — arstack. Function EvElem returns the descriptor value for the ith element in an array. Detailed descriptions for these functions can be found early in Chapters 3 and 4. Whenever an E_Derefa event occurs, the variable being accessed is pushed onto the stack. Upon receiving an E_Index event, the variable on the top of the stack is popped off. The range of the referencing index for this variable is then checked using function EvElem. This function returns a NULL descriptor if the index is not valid.

Chapter 6

Performance

This chapter discusses the performance of the Alamo C framework. Due to frequent context switches between the TP and EMs and the fact that EMs take time processing each event, monitoring imposes significant overhead on the TP's execution. The actual slow down of the program execution depends on the frequency of the events and the intensity of an EM's response to those events. The more complicated the analysis an EM needs to do, the more overall time spent in the monitor. However, most EMs are developed to monitor a particular aspect of the TP's execution. Execution monitors are often simple programs that do not require intensive analysis.

Performance results for the Alamo C framework are preliminary. There are three primary performance issues. The first one is the cost of the test performed in the instrumentation in order to decide whether to report an event or not. The second one is the cost of the context switches between target program and monitors when events are reported. The third one is the cost of memory protection which protects the memory that belongs to monitors when the target program gets the control. There is also a cost associated with loading each program into the shared address space and relocating symbols for them. This cost is not considered here because it is a one time expense at program load-time which varies with program size and is typically the order of 0.1 second. At the end of this chapter, the overall slowdown of the target program execution under several situations is discussed. All the tests were run on a 167 MHz UltraSPARC workstation.

6.1 Cost of event mask testing

The framework decides what kind of events can be instrumented, and the user decides what kind of events he/she needs. Not every event is reported to the monitor. If instrumentation is enabled for many or all events, most of the tests for deciding whether or not to report an event fail. The worst-case slowdown due to event mask tests can be measured by letting a fully instrumented target program execute with an empty event mask. Since the target program has to be compiled with the **-g** option for obtaining its debugging symbol table information, it cannot be compiled with the compiler optimization flag **-O** on.

In order minimize the cost of the event mask testing, the Sparc code for it has to be smart enough to take the advantage of the fact that the event code is a manifest constant. From this code, both the word within the mask to use and the bit to check are known at compile time. The EV() macro is tuned to compute which word within the event mask to test and the word mask for that word at compile time instead of doing the computing work at run-time. In this way the testing is only an array reference operation for loading that particular word followed by a bit AND operation with that word mask. On the UltraSPARC, each such test operation takes on the order of 0.1 μs to finish.

6.2 Cost of context switch

The worst-case time spent for a context switch between target program and monitors is measured by timing the slowdown of the target program execution when all events are reported, but the monitor does nothing with the events. It is the order of 8 μs using the coroutine approach and 40 μs using the Solaris thread model. Generally monitors add their own substantial slowdown to the equation for processing events. Comparing the Solaris thread model with the coroutine model is unfair in the sense that semaphore operations are very expensive. The timing difference between these two models will be less if mutex locks and conditional variables were used for the synchronization of multiple threads. The overhead associated with the coroutine model is small since the scheduling and context switch are explicit.

6.3 Cost of memory protection

As mentioned in Chapter 2, the overall impact of the memory protection on the execution speed depends upon the size of the memory that needs protection. For example, using the simple floating point operation profiling monitor, the size of memory pages that need to be protected is the order of 167 Kbyte which includes the heap space for the monitors. (The heap space allocated for each monitor varies from program to program.) In this example, each memory protection operation takes on the order of 130 μs time to finish.

6.4 Overall effect of monitoring upon execution speed

Table 6.1 shows the overall execution time of three target programs running in the following situations:

- target programs compiled with -O flag run alone with no instrumentation

	-O	-g	instrumented	event reports	memory protection
laplace.c	0.62	0.68	1.42	137.54	2130.15
life.c	0.22	0.26	1.52	143.66	2160.37
concord.c	0.53	0.58	0.62	1.81	15.58

Table 6.1: Execution time (in sec) for three target programs in five different cases, running on a 167 MHz UltraSPARC with 64 MB RAM.

- target programs compiled with -g flag run alone with no instrumentation
- instrumented target programs run without monitors
- instrumented target programs report all events to a monitor. The monitor increments its counter each time an event report occurs.
- instrumented target programs report all events to a monitor with memory protection feature enabled.

The target programs are laplace.c, life.c, and concord.c. *laplace.c* is a program simulating the solution to Laplace's equation at a point in an annulus via Monte Carlo method. The solution is approximated using 1000 random walks. *life.c* is Conway's Game of Life; executed for 30 generations. *concord.c* is a program producing a concordance showing the lines on which each word occurs within a text file. Concord is run on a 110 Kbyte input file. In the current implementation, instrumented target programs have to run with a monitor coordinator even though there are no additional monitors. Also a monitor coordinator is loaded even if there is only one monitor monitoring the target program. The implementation for the context switch between the TP and EMs uses the coroutine approach. TP and EMs have their own heap spaces.

Each column in Table 6.1 represents one case for a target program in the order listed above. The five columns introduce cumulative, successively more expensive elements of monitoring overhead. In the third column, Alamo's first source of overhead is introduced by the event mask testing, although the mask is the empty set and no events are reported. The numbers of event mask tests performed are 6939810, 7157416, and 57821 for laplace.c, life.c, and concord.c, respectively. The first two cases illustrate exhaustive instrumentation of all available events, while the last case illustrates a more selective instrumentation configuration. The fourth column introduces overhead from context switches, since every event test succeeds and the actual event report occurs. The number of event reports is the same as the number of event mask tests in the third column for all three target programs. In the fifth column, besides the sources of overhead mentioned for the fourth column, there is an additional overhead associated with the memory protection function calls. For all three target programs, the size of the memory protected is 167 Kbyte, since they use the same monitors.

Comparing the execution times in columns 1 and 2 for all three target programs, it is evident that compiler optimization does not affect these programs' performance much. A similar observation can be made in comparing columns 2 and 3; instrumentation does not cost too much in terms of execution speed.

The cost associated with the instrumentation of the program depends on the original target program execution time and the actual number of the event mask tests. Although different instrumented programs have different number of event mask tests, most of them are in the same order. The cost of instrumentation will vary most in the following two extreme cases: the first case is for a target program which is heavily CPU bound, and the second case is for a target program which is heavily I/O bound. The instrumentation will have far more impact on the overall target program execution in the first case than in the second case.

Comparing the execution times in columns 3, 4, and 5, there are big jumps in the timing. The slowdown of the target program execution can vary from only 3 times to almost 100 times depending on the number of event reports. The worst-case slowdown of the target program execution is significant when there are millions of event reports. On top of that, the memory protection feature can further slow down the program execution more than 10 times in the worst-case. However, memory protection feature is still affordable if it is not enabled for every single event report. The measure of worst-case slowdown due to the context switch simply illustrates the entire motivation for event masking, which is to reduce the number of context switches. In addition, with the configurability of the instrumentation, higher-level events can be generated. This allows monitor writers to ask for fewer events but still can obtain the same amount of information they need.

Chapter 7

Conclusions and Future Work

The Alamo framework's design is successful in extending the earlier work developed for an interpretive language by developing techniques suitable for the compiled systems programming language ANSI C. The implementation of the Alamo monitor executive provides an execution environment in which users can develop multiple EMs to monitor a C program's execution quite rapidly. The Alamo framework also provides EM writers with direct easy access to the TP's state and variable information. The successful implementation of the framework demonstrates that Alamo's design is feasible for applications that monitor the execution of ANSI C programs.

This chapter first summarizes what was implemented, followed by the current limitations of the framework. It also discusses desirable enhancements and future work.

7.1 Contributions of the AME

The major contribution of this work is the successful implementation of the synchronous thread execution model, the context switch between the TP and EMs, the dynamic code loading facility, and the TP access library routines for C programs compiled into relocatable ELF object files. The following paragraphs present a summary of these contributions.

Synchronous thread execution model — AME implements the synchronous thread execution model. Because in the thread model, EM and TP are two different entities but they share the same address space, context switch time between them is much less than that between two different processes and accessing the TP's information is much easier for EMs. Since the TP and EMs run sequentially instead of concurrently, this gives EMs ability to obtain full stable information about the behavior of the TP and equally importantly, it substantially simplifies monitor development.

Event-driven control of the target program — The execution of the TP is driven by events. Whenever a desired event occurs in the TP, execution control is transferred to the EM along with a code and a value for that event. When an EM resumes the execution of the TP, it uses an event mask to explicitly specify what kind of events are to be reported in the execution that follows. Thus EMs can dynamically change the event mask in between event reports. In addition, events are masked in the TP instead of filtering them in the EM. This reduces the number of context switches between the TP and EMs. The primary test for whether an event is of interest is performed in-line with no context switch to the monitor.

Dynamic loading without linking — The dynamic code loading facility gives users the flexibility to choose what tools to use and how many of them to use. This code loading facility does not link variables in different loaded programs. Each loaded program is linked by itself and the only remaining external references are shared library function calls. AME can provide a separate heap space for each loaded program. Memory protection can be provided on both EM code and data regions.

Easy direct target program access — EMs can gather extensive information from the TP, starting with the information provided from instrumented events. To acquire further information, EMs can use a set of functions provided in the TP access library to access information about TP symbol tables, scopes, addresses and values of variables. This functionality enables the EM writer to extract more information from the running target program and communicate with the TP more efficiently.

Coordination and synchronization using MCs — The dynamic loading capability allows multiple EMs to be loaded to monitor a single TP. This allows users to write simple special purpose EMs quite rapidly. Besides the difficulty posed by loading the programs, there is additional difficulty coordinating and transferring control among those monitors. MCs are used to perform synchronization and provide monitoring services to additional EMs. With a MC, EMs do not need to be aware of one another and of the MC itself. This in turn simplifies the implementation of EMs.

7.2 Limitations of the AME

The first limitation of the AME is that the code loading facility supports only systems using the ELF object format. This limits our framework's portability. All the work has been done so far under Solaris 2.x. The ELF code loader was borrowed from the Linux kernel, so we are optimistic that at least it can be ported there.

The second limitation is caused by the intrusion problem in our framework. When the act of monitoring a program changes the behavior under observation, it is called *intrusion*. Intrusion is a common problem in any execution monitoring system. There are some aspects of the intrusion problem in our framework. Since the TP is instrumented for event generation, its memory layout is altered. Changing the memory layout of a program may result in hiding certain memory bugs with random pointers or it may cause them to surface. This kind of source code intrusion problem limits the usefulness of our framework when monitoring programs which have incorrect behavior due to memory bugs. The source-language level instrumentation also inserts additional instructions and temporary variables into the TP. However this kind of code intrusion doesn't affect the monitoring task as long as the information required by EMs is not at the machine-language level, such as the offsets for variables on the stack, which registers contain which variables, *etc.* It is important to point out that the dynamic loading facility implemented in our framework avoids certain code intrusion problems in the sense that programs are loaded into disjoint regions in memory and no linking procedure is done on the variables used by monitors with those used by the target program. In addition, data intrusion is avoided by using separate global sections, heaps, and stacks for the loaded programs. Intrusion can also refer to the execution slowdown imposed by monitoring. In real-time and concurrent systems this can render monitoring useless. Since research done on monitoring sequential programs forms the basis on which to develop monitors for parallel programs, our framework targets monitoring sequential program execution behavior. Temporal intrusion is not considered in this work. The effect of monitoring on execution speed is considered only so far as to establish framework usability on "real" C programs.

The third limitation is caused by the execution model we chose for the AME. The current framework supports monitoring sequential programs. Since the execution model uses a synchronous thread model, it is not suitable for monitoring shared memory preemptive parallel programs and real-time applications. The reason is that in a shared memory multiprocessor system, when a thread of the TP running on one processor is suspended by an EM, a thread of the TP running on another processor can change some portion of the shared memory at the same time as the EM is accessing them. This is not desirable for the EM which is gathering the TP's state information. However, this framework can be extended to monitor distributed programs and user-level non-preemptive multi-threaded programs, since in a distributed system, programs running on separate machines do not share memory. An EM can be attached to each program running on a separate machine. It is up to EM writers to develop EMs which can communicate with each other. For a user-level non-preemptive multi-threaded program, an EM can be attached to each thread. Since it is non-preemptive, a thread's control won't be taken over by another thread unless it explicitly relinquishes its CPU time to another thread. Thus the EM can access the TP state without being concerned that the TP's state can be modified by another thread. For real-time applications, causing the real-time target program to stop its execution for arbitrary lengths of time while EMs process the information is not suitable. Even if such monitors themselves are written to guarantee real-time performance, the presence of any number of such monitors in the typical scenario advocated by this framework prevents the target program from meeting its real-time constraints.

7.3 Future work

The execution monitoring framework for ANSI C programs was motivated by a desire to develop new types of execution monitors, particularly program visualization tools. One future work is to demonstrate the feasibility of this framework by developing sophisticated execution monitors. Another future project is to port this framework to other operating systems that use ELF object format. Portability would increase the value of the framework. In addition, development of advanced execution monitoring tools requires the ability to easily prototype them in a very high-level language, and then migrate completed tools to a compiled language. Future research work can be done to facilitate this process by building a bridge between the current framework and its predecessor framework to support multi-lingual program execution monitoring. Yet another future work would be to extend this framework to monitor distributed programs and multi-threaded programs. This requires a substantial work in coordinating and synchronizing the execution of TP and monitors.

Appendix A

This appendix presents EM library functions introduced in Chapter 4. It also presents examples of using these functions.

A.1 List of EM library functions

- int EvSize(Desc var);

  This function returns the size of a variable, similar to sizeof in C.

- char * EvTypeString(Desc var);

  This function returns a string description of the type of the variable. For a structure or union type variable, it gives out each field's type but is restricted to one level.

- int EvType(Desc var);

  This function returns an integer value representing one of the predefined C variable types. The predefined variable types are: INT, CHAR, LONG_INT, UNSIGNED_INT, SHORT_INT, LONG_UNSIGNED_INT, SHORT_UNSIGNED_INT, SIGNED_CHAR, UNSIGNED_CHAR, LONG_DOUBLE, FLOAT, DOUBLE, VOID, STRUCT, UNION, ENUM, POINTER, FUNCTION, ARRAY.

- long Derefl(Desc var);

  This function returns a long for the variable of types CHAR, UNSIGNED_CHAR, SIGNED_CHAR, INT, LONG_INT, UNSIGNED_INT, LONG_UNSIGNED_INT, SHORT_INT, SHORT_UNSIGNED_INT. We promote 1-byte and 2-byte integer values to 4-byte values. It is up to the EM writers to type cast it to the desired type.

- long double DerefD(Desc var);

  This function returns a long double value for the variable of types LONG_DOUBLE, DOUBLE, FLOAT. We promote 4-byte and 8-byte floating point values to 16-byte values. The EM authors can type cast it to a float or double.

- Desc DerefP(Desc var, int i);

  This function returns a descriptor with its type field being the type of what the pointer points to and its addr field being the address where the ith object of this pointer points to is stored.

- char *EvImage(Desc var);

  This function returns a string image for the value of a variable. It only works for the variable types of INT, CHAR, LONG_INT, UNSIGNED_INT, LONG_UNSIGNED_INT, SHORT_INT, SHORT_UNSIGNED_INT, SIGNED_CHAR, UNSIGNED_CHAR, DOUBLE, LONG_DOUBLE, FLOAT.

- void *EvAddr(Desc var);

  This function returns the address of the variable specified.

- void EvLoc(char *filename, int *line_num);

  This function does a lookup using the internal program counter and frame pointer values, and writes the filename and current line number information into the parameters given.

- int IsMember(Desc var1, char *var2);

  If var1 is of type structure or union, tests whether var2 is a field of var1.

  If var1 is of type function, tests whether var2 is a local variable of var1.

- int IsNull(Desc var);

  Tests whether a descriptor's addr and type fields are NULL.

- int IsSameType(Desc var1, Desc var2);

  Tests if var1 and var2 have the same type fields.

- int IsEQ(Desc var1, Desc var2);

  This function first calls IsSame to test the types of var1 and var2. If it is false, returns -1 indicating error. Otherwise, tests if var1 and var2 have the same addr fields, returns 1 for true, zero for false.

- int IsGT(Desc var1, Desc var2);

  This function first calls IsSame to test if var1 and var2 have the same type. If it is false, returns -1 indicating error. Otherwise, tests if var1's value is strictly greater than var2's value, returns 1 for true, zero for false.

- int IsGE(Desc var1, Desc var2);

  This function first calls IsSame to test if var1 and var2 have the same type. If it is false, returns -1 indicating error. Otherwise, tests if var1's value is greater than or equal to var2's value, returns 1 for true, zero for false.

- int IsLE(Desc var1, Desc var2);

  This function first calls IsSame to test if var1 and var2 have the same type. If it is false, returns -1 indicating error. Otherwise, tests if var1's value is less than or equal to var2's value, returns 1 for true, zero for false.

- int IsLT(Desc var1, Desc var2);

  This function first calls IsSame to test if var1 and var2 have the same type. If it is false, returns -1 indicating error. Otherwise, tests if var1's value is strictly less than var2's value, returns 1 for true, zero for false.

A.2 Examples of using these functions

A.2.1 Array element traversing

An example of traversing fields of a structure and indexing elements inside an array. Print out the value of the fields or elements in the array if they are of simple types:

```
Desc d, temp;
int i, j, typecode;
char *s;
...
for ( i = 0; i < EvNum(Locals); i ++ ) {
    d = EvElem(Locals, i);
    if ( EvType(d) == STRUCT || EvType(d) == ARRAY ) {
        for ( j = 0; j < EvNum(d); j ++ ) {
            ...
             /*
              * get the jth field's or element's variable name
              */
            s = EvName(d, j);
             /*
              * get the jth field's or element's descriptor
              */
            temp = EvElem(d, j);
             /*
              * Or using EvVar(s) to get the jth field's or
              * element's descriptor.
              * temp = EvVar(s);
              */
             /*
              * get the jth field's variable type
              */
            typecode = EvType(temp);
            switch (typecode) {
                case INT:
                        printf("integer value: %d \n", DerefI(temp));
                        break;
                case CHAR:
                        printf("char value: %c \n", DerefC(temp));
                        break;
                case DOUBLE:
                    printf(”double value: %f \n”, DerefD(temp));
                    break;
                case FLOAT:
                    printf(”float value: %f \n”, DerefF(temp));
                    break;
                /*
                 * Or we can use EvImage() to get the string image
                 * of the value for a variable of type INT, CHAR,
                 * DOUBLE, FLOAT, etc.
                 * printf(”var value: %s \n”, EvImage(temp));
                 */
                 ...
            }
            ...
        }
    }
}
```

A.2.2 Linked list traversing

An example of traversing a linked list. In C, the statement like: for (p = x; p; p = p->next) is typical. An EM writer can write similar statements for the same purpose. The difference is the EM writer usually doesn’t know the structure of variable “x”. In order for him to do the same thing, he has to do a little more work to find out the “next” field for this structure. The code sample would be:

```
Desc d;
int i, index;
/*
 * find out the ”next” field in the structure,
 * assign that element’s index to variable index
 */
d = EvVar(”x”);
for ( i = 0; i < EvNum(d); i ++ ) {
    if ( IsSameType(d, EvElem(d, i)) ) {
        index = i;
        break;
    }
}
```

```
for ( d = EvVar(”x”); !IsNull(d); d = EvElem(d, index) ) {
    ...
}
```

A.2.3 Variables in different scopes

An example of querying information about a variable which is not in the current scope.

```
char *s;
int i, j;
Desc d;
...
for ( i = 0; i < EvNum(Locals); i++ ) {
    s = EvName(Locals, i);
    for ( j = 1, d = WhereIs(s, j); !IsNull(d);
           j++, d = WhereIs(s, j) ) {
          ...
    }
    ...
}
```

A.2.4 Stack frame traversing

An example of traversing the stack frame of TP. Print out the filename and line number information for the TP.

```
char *filename;
int line_num;
...
/*
 * get current filename and line number info
 */
EvLoc(filename, &line_num);
printf(”filename: %s, line_number: %d \n”, filename, line_num);
while ( UpStack() != −1 ) {
    ...
    EvLoc(filename, &line_num);
    printf(”filename: %s, line_number: %d \n”, filename, line_num);
}
```

Appendix B:

This appendix presents the grammar for the .stabstr section generated by GNU's C compiler with **-g** command line option. It is a simplified grammar for decoding the stabs string section based on materials presented in [Stab93].

```
stab: STRING COLON symbol_des type_info
        | STRING COLON type_info
        | COLON symbol_des type_info
        | COLON type_info
symbol_des: f | F | G | p | P | T | t | r
type_info: INT
          | INT EQ type_description
type_description: range_des SM
                 | pointer_des
                 | array_des
                 | struct_union_des SM
                 | enum_des SM
                 | funtion_des
                 | crossref_des
crossref_des: x s STRING COLON
             | x u STRING COLON
             | x e STRING COLON
range_des: r type_info SM INT SM INT
array_des: a index_type SM type_info
index_type: INT
           | range_des
pointer_des: ASTERISK type_info
struct_union_des: s INT struct_union_lists
                 | u INT struct_union_lists
struct_union_lists: struct_union_list SM
                   | struct_union_list SM struct_union_lists
struct_union_list: STRING COLON type_info CM INT CM INT
enum_des: e enum_lists
enum_lists: STRING COLON INT CM
           | STRING COLON INT CM enum_lists
function_des: f type_info
```

Appendix C:

This appendix lists the basis set of event types that are currently supported. In addition, the names of common event families are supplied. Families are names for sets of related events; the values associated with a given family are the values associated with the specific events in that family.

Higher level events for library-defined behavior (such as I/O) are not part of the basis, but are instead defined by configuration directives. See [Temp96] for a description of the configurable C instrumentation tool.

event code	**event value**	**description**
E_Arith	family	arithmetic operators
E_Addi	value	integer addition
E_Addf	value	float addition
E_Addd	value	double addition
E_Addc	value	character addition
E_Subi	value	integer subtraction
E_Subf	value	float subtraction
E_Subd	value	double substraction
E_Subc	value	character substraction
E_Muli	value	integer multiply
E_Mulf	value	float multiply
E_Muld	value	double multiply
E_Mulc	value	character multiply
E_Divi	value	integer division
E_Divf	value	float division
E_Divd	value	double division
E_Mod	value	integer mod
E_Neg	value	unary negate

E_Assign	family	assignment
E_Lvaluei	address	integer lvalue assigned
E_Lvaluef	address	float lvalue assigned
E_Lvalued	address	double lvalue assigned
E_Lvaluec	address	character lvalue assigned
E_Lvaluep	address	pointer lvalue assigned
E_Lvalues	address	structure lvalue assigned
E_Assigni	value	integer assignment
E_Assignf	value	float assignment
E_Assignd	value	double assignment
E_Assignc	value	character assignment
E_Assignp	value	pointer assignment
E_Assigns	value	structure assignment
E_MulAssigni	value	integer multiply assignment
E_MulAssignf	value	float multiply assignment
E_MulAssignd	value	double multiply assignment
E_MulAssignc	value	character multiply assignment
E_MulAssignp	value	pointer multiply assignment
E_DivAssigni	value	integer division assignment
E_DivAssignf	value	float division assignment
E_DivAssignd	value	double division assignment
E_ModAssign	value	mod assignment
E_AddAssigni	value	integer addition assignment
E_AddAssignf	value	float addition assignment
E_AddAssignd	value	double addition assignment
E_AddAssignc	value	character addition assignment
E_SubAssigni	value	integer subtraction assignment

E_SubAssignf	value	float subtraction assignment
E_SubAssignd	value	double subtraction assignment
E_SubAssignc	value	character subtraction assignment
E_ShiftlAssign	value	shift left assignment
E_ShiftrAssign	value	shift right assignment
E_BitAndAssign	value	bit AND assignment
E_BitXorAssign	value	bit XOR assignment
E_BitOrAssign	value	bit OR assignment
E_AndAssign	value	logical AND assignment
E_OrAssign	value	logical OR assignment
E_Logical	family	logic operators
E_Or	value	logical OR
E_And	value	logical AND
E_Eq	value	logical EQUAL
E_Neq	value	logical NOT EQUAL
E_Less	value	logical LESS THAN
E_Lesseq	value	logical LESS THAN EQUAL
E_Grt	value	logical GREATER THAN
E_Grteq	value	logical GREATER THAN EQUAL
E_Bang	value	logical NOT
E_Bit	family	bitwise operators
E_BitOr	value	bitwise OR
E_BitAnd	value	bitwise AND
E_BitXor	value	bitwise XOR
E_BitNot	value	bitwise complement
E_Shiftl	value	shift left

E_Shiftr	value	shift right
E_Ref	family	memory (de)references
E_Refi	address	dereference integer
E_Reff	address	dereference float
E_Refd	address	dereference double
E_Refc	address	dereference character
E_Refa	address	dereference array
E_Refp	address	dereference pointer
E_Refs	address	dereference structure
E_Reffn	address	dereference function
E_IndexS	value	structure index (field access)
E_Index	value	array index
E_Addrof	address	address of operator
E_Cont	value	content of operator
E_Sizeof	value	size of operator
E_Postinc	value	post−increment operator
E_Postdec	value	post−decrement operator
E_Preinc	value	pre−ncrement operator
E_Predec	value	pre−ecrement operator
E_Proc	family	procedure activity
E_Pcall	address	procedure call
E_Pret	value	procedure return value
E_Pparam	value	procedure parameters
E_Fcall	address	library function call
E_Fret	value	library function return value
E_Fparam	value	library function parameters

E_Control	family	control flow activity
E_If	value	If test
E_Cond	value	Conditional operator test (? :)
E_Loop	value	Loop test (for, while)
E_Switch	value	Switch test
E_Continue	(none)	Continue statement
E_Break	(none)	Break statement
E_Label	string	Destination (switch branch, goto)

Bibliography

[Aral88] Z. Aral and I. Gertner, Non-intrusive and Interactive Profiling in Parasight, *Proceedings of the ACM/SIGPLAN PPEALS 1988*, 21-30, Sept. 1988.

[Brow84] M. H. Brown and R. Sedgewick, A System for Algorithm Animation, *Computer Graphics*, **18**(3), 177-186, July 1984.

[Brue90] B. Bruegee, BEE: A Basis for Distributed Event Environments (Reference Manual), CMU-CS-90-180, Carnegie-Meellon University, Nov. 1990.

[Brue93] B. Bruegge, T. Gottschalk, and B. Luo, A Framework for Dynamic Program Analyzers, *OOPSLA '93 Proceedings, Sigplan Notices*, 28(10), 65-82, Oct. 1993.

[Elf93] Linker and Libraries Manual, *Solaris 2.3 Answerbook*, Sun Microsystem, November, 1993.

[Fras91] C. W. Fraser and D. R. Hanson, A Retargetable Complier for ANSI C, *SIGPLAN Notices*, **26**(10), 29-43, Oct. 1991.

[Gris90] R. E. Griswold and M. T. Griswold, *The Icon programming language*, Prentice Hall, 1990.

[Hans96] D. R. Hanson and M. Raghavachari, A Machine-Independent Debugger, *Software-Practice and Experience*, Vol. **26**(7), 1-24, July 1996.

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

[Ho91] W. Wilson Ho and R. A. Olsson, An Approach to Genuine Dynamic Linking, *Software Practice and Experience*, Vol. **21**(4), 375-390, April 1991.

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

[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.

[Laru95] J. R. Larus and E. Schnarr, EEL: Machine-Independent Executable Editing, *SIGPLAN Conference on Programming Language Design and Implementation*, June, 1995.

[Marl80] C. Marlin, *Coroutines (Lecture Notes in Computer Science 95)*, Springer-Verlag, Berlin, 1980.

[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.

[Plat81] B. Plattner and J. Nievergelt, Monitoring Program Execution: A Survey, *IEEE Computer*, pp. 76-93, 1981.

[Sosi1] R. Sosic, The Dynascope Directing Server: Design and Implementation, *USENIX Association, Computing Systems*, Vol. **8**(2), 107-133, 1995.

[Sosi2] R. Sosic, A Procedural Interface for Program Directing, *Software Practice and Experience*, Vol. **25**(7), 767-787, July 1995.

[Sriv93] A. Srivastava and D. W. Wall, A Practical System for Intermodule Code Optimization at Link-Time, *Journal of Programming Language*, **1**(1), 1-18, March 1993.

[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.

[Stab93] J. Menapace, J. Kingdon, and D. MacKenzie, The "stabs" Debug Format, *Cygnus Support*, 1993.

[Temp96] K. S. Templer and C. L. Jeffery, Design of a Configurable C Instrumentation Tool, *Technical Report*, **96-6**, Division of Computer Science, University of Texas at San Antonio, 1996.

[Wamp81] Stephen B. Wampler, Control Mechanisms for Generators in Icon, *Technical Report*, **81-18**, Department of Computer Science, University of Arizona, 1981.

[Zhou96] W. Zhou and C. L. Jeffery, Target Program State Access in the Alamo Monitor Framework, *Technical Report*, **96-5**, Division of Computer Science, University of Texas at San Antonio, 1996.

Vita

Wenyi Zhou was born on October 3, 1969 in Shanghai, China. She received a B.S. degree in Materials Science from the Shanghai Jiao Tong University in China in 1991, and a M.S. degree in Materials Science from the State University of New York at Stony Brook in 1993. Her current interests include execution monitoring, program visualization, operating systems research and development, and internetworking.

Permanent address:	196-12, 47th Ave.
	Flushing, NY 11358

This thesis was typeset by the author in LaTeX.